NICOLAS HILAIRE

APPRENEZ À DÉVELOPPER EN C#

LA PROGRAMMATION EN C#.NET EXPLIQUÉE AUX DÉBUTANTS !

PRÉFACE D'ÉRIC MITTELETTE
Responsable des relations techniques avec les développeurs chez Microsoft France





www.siteduzero.com

NICOLAS HILAIRE

APPRENEZ À DÉVELOPPER EN C#



LA PROGRAMMATION EN C#.NET EXPLIQUÉE AUX DÉBUTANTS !

PRÉFACE D'ÉRIC MITTELETTE
Responsable des relations techniques avec les développeurs chez Microsoft France



www.siteduzero.com

DANS LA MÊME COLLECTION



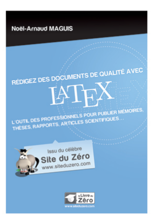
CONCEVEZ VOTRE SITE WEB AVEC PHP ET MYSQL

MATHIEU NEBRA
ISBN : 978-2-9535278-1-0



REPRENEZ LE CONTRÔLE À L'AIDE DE LINUX

MATHIEU NEBRA
ISBN : 978-2-9535278-2-7



RÉDIGEZ DES DOCUMENTS DE QUALITÉ AVEC LATEX

NOËL-ARNAUD MAGUIS
ISBN : 978-2-9535278-4-1



APPRENEZ À PROGRAMMER EN JAVA

CYRILLE HERBY
ISBN : 978-2-9535278-3-4



PROGRAMMEZ AVEC LE LANGAGE C++

M.NEBRA ET M.SCHALLER
ISBN : 978-2-9535278-5-8



RÉDIGEZ FACILEMENT DES DOCUMENTS AVEC WORD

MICHEL MARTIN
ISBN : 978-2-9535278-7-2



APPRENEZ À PROGRAMMER EN PYTHON

VINCENT LE GOFF
ISBN : 979-10-90085-03-9



DÉBUTEZ DANS LA 3D AVEC BLENDER

ANTOINE VEYRAT
ISBN : 978-2-9535278-9-6



RÉALISEZ VOTRE SITE WEB AVEC HTML5 ET CSS3

MATHIEU NEBRA
ISBN : 978-2-9535278-8-9



APPRENEZ À PROGRAMMER EN C • 2e ÉDITION

MATHIEU NEBRA
ISBN : 979-10-90085-00-8

NICOLAS HILAIRE



LA PROGRAMMATION EN C#.NET EXPLIQUÉE AUX DÉBUTANTS !

PRÉFACE D'ÉRIC MITTELETTE
Responsable des relations techniques avec les développeurs chez Microsoft France



www.siteduzero.com



Simple IT 2012 - ISBN : 978-2-9535278-6-5

Préface

C# .NET est **le** langage de programmation phare de Microsoft. Il a été développé dans le but de pouvoir créer *facilement* divers types d'applications en tirant le meilleur des produits et technologies Microsoft.

Les créateurs du langage se sont inspirés des langages existants en s'attachant à retenir le meilleur de chacun d'eux. Aussi n'est-ce pas étonnant de retrouver un typage fort, une approche orientée objet et une syntaxe rappellant à la fois celle du C++ et du Java. C# .NET est apparu en 2000 et depuis, ne cesse d'évoluer au rythme des différentes versions du Framework .NET. Le couple C# et Framework .NET englobe les dernières avancées des langages de programmation (Generic, Lambda, Inférence de type, Linq...). Ces améliorations, fortement inspirées des langages dits fonctionnels, font de C# un des langages les plus modernes et aboutis, sans que jamais la productivité et la solidité du code ne soient compromis. Aujourd'hui, C# .NET est de plus en plus utilisé dans le monde professionnel. Sa puissance et son interopérabilité avec les produits et technologies Microsoft font de lui un langage sûr et pérenne. Ce langage présente en outre l'intérêt de ne pas être propriétaire puisque ses spécifications permettent de voir apparaître des initiatives (comme par exemple Mono), le code C# pouvant ainsi tourner sur des distributions Linux. Il est possible de développer toutes sortes d'applications : jeux, applications de gestion, interfaces tactiles, XAML ou applications pour téléphones. N'oublions pas le monde embarqué avec le Micro Framework .NET ainsi que le Web avec ASP.NET. Bref, C# est un langage tout terrain, ouvrant une gamme de possibles unique sur la plateforme Microsoft.

Ce livre se veut simple et facile d'accès. Il allie les connaissances de Nicolas Hilaire, spécialiste de ces technologies et MVP Microsoft [1], avec celles des créateurs du Site du Zéro, réputés depuis de nombreuses années pour leur approche pédagogique et accessible à tous les débutants. Ce livre est donc tout indiqué pour ceux qui veulent se former facilement à la programmation C# .NET.

Éric Mittelette
Responsable des relations techniques avec les développeurs chez Microsoft France

1. *Most Valuable Professional*, expert en technologies Microsoft.

Avant-propos

Quand j'ai commencé la programmation, j'avais dix ans et un Atari ST possédant un interpréteur GFA Basic. Mes parents m'avaient acheté un livre contenant des listings à recopier et à exécuter. Si mes souvenirs ne me trahissent pas, il s'agissait pour la plupart d'applications permettant de gérer le contenu de son frigo ou de sa cave à vins. Quelques petits jeux très simples et peu graphiques venaient agrémenter le lot. Pour faire fonctionner ces programmes, il fallait tout recopier à la main (ou plutôt au clavier), généralement quelques centaines de lignes de code. Régulièrement, cela ne fonctionnait pas car je faisais une erreur de copie, inversant des parenthèses ou oubliant des mots. À part vérifier tout le listing ligne par ligne, je n'avais plus qu'à passer au listing suivant ! Parfois, mes efforts étaient récompensés même si je ne comprenais strictement rien à ce que je recopiais. Je me rappelle d'un superbe labyrinthe en 3 dimensions, quoique mes souvenirs lui rendent certainement un hommage plus en couleur qu'il ne le méritait ! Ces listings remplis de mots magiques m'ont donné envie de comprendre comment cela fonctionnait. J'ai donc pris mon courage à dix doigts et tenté de créer mes propres programmes en isolant les parties qui me paraissaient simples. Afficher « Bonjour comment allez-vous » et pouvoir « discuter » avec l'ordinateur grâce à un algorithme de mon cru ont été un de mes premiers souvenirs de programme réussi.

À cette époque reculée, il n'existait pas de moyen d'apprendre facilement la programmation. Il n'y avait pas internet... eh oui, cette époque a existé ! Durant mon adolescence j'ai continué mon apprentissage en essayant différents langages, comme le C++ ou l'assembleur, le turbo pascal et autres joyeusetés. La plupart étaient inaccessibles, notamment le C++. Quelques livres en bibliothèque ont fini dans la mienne mais ils étaient tous bien incompréhensibles... je me souviens même d'un livre qui promettait de pouvoir créer un jeu « facilement ». Cela ne devait pas être si facile que ça vu mon air hébété après la lecture du livre ! Cela manquait d'un Site du Zéro où tout est expliqué de zéro pour les personnes, comme j'ai pu l'être, curieuses de se lancer dans le monde magique du développement.

On parle du C# ?

J'y viens ! C'est dans cette optique que j'ai commencé à écrire. Pouvoir partager mes connaissances souvent durement acquises et aider ceux qui ont du mal à se lancer. Et c'est vrai que ce n'est pas facile, malgré toute la bonne volonté du monde. Sans une méthodologie simple et des explications claires, il n'est pas aisé de se lancer sans se sentir perdu. C'est là où j'espère pouvoir faire quelque chose à travers la collection des Livres du Zéro. Après tous mes essais de jeunesse, mes études et mon entrée dans le monde du travail, j'ai acquis une certaine expérience des différents langages de programmation. J'ai pris goût à l'écriture en commençant à rédiger des articles avec mon langage préféré de l'époque, le C++. Aujourd'hui, c'est le C# qui occupe cette place prestigieuse dans mon classement ultra-personnel des langages de programmation ! C'est donc l'occasion de pouvoir mettre à profit cette volonté de partage de connaissances et ce goût pour la rédaction, dans un ouvrage permettant d'apprendre le C# et qui est destiné aux débutants.

Qu'allez-vous apprendre en lisant ce livre ?

Nous allons apprendre le langage de programmation C# de façon progressive au cours de cet ouvrage, composé des parties suivantes :

1. **Les rudiments du langage C#** : nous commencerons par découvrir les bases du langage C#. Nous partons vraiment des bases : comment est construite une application informatique ? Quels logiciels dois-je installer ? Quelles sont les instructions de base du C# ? Nous allons découvrir tout cela au cours de cette première partie qui permettra de poser les briques de nos premières applications.
2. **Un peu plus loin avec le C#** : dans cette partie, nous allons continuer à approfondir nos connaissances avec le C#. Nous découvrirons les premières interactions avec l'utilisateur de nos programmes. Comment lire simplement une saisie clavier ? Comment lire le contenu de la ligne de commande ? Nous découvrirons cela, avec en complément des TP pour nous entraîner.
3. **Le C#, un langage orienté objet** : ici, les choses sérieuses commencent et nous allons voir ce qu'est la programmation orientée objet et comment le C# répond à ce genre de programmation. Chapitre un peu plus avancé où vous découvrirez toute la puissance du langage et où vous vous rendrez compte de l'étendue des possibilités du C# !
4. **C# Avancé** : forts de nos connaissances acquises précédemment, nous étudierons des points plus avancés dans cet ultime chapitre. Nous verrons comment accéder efficacement aux données grâce à LINQ et comment utiliser une base de données avec Entity Framework. Nous verrons également d'autres aspects permettant d'être encore plus efficaces avec vos développements.

À la fin de cet ouvrage, vous aurez acquis toutes les bases vous permettant de vous lancer sans appréhension dans le monde du développement d'applications professionnelles avec le C#. Vous découvrirez en bonus un aperçu des différentes applications que l'on peut réaliser avec le C#.

Comment lire ce livre ?

Esprit du livre

Oui, oui, vous avez bien lu, ce livre est pour les débutants. Pas besoin d'avoir fait du développement auparavant pour pouvoir lire cet ouvrage ! Je vais donc faire de mon mieux pour détailler au maximum mes explications, c'est promis. Bien sûr, il y en a peut-être parmi vous qui ont déjà fait du C, du C++, du Java. . . Évidemment, si vous avez déjà fait du développement informatique, ce sera plus facile pour vous. Attention néanmoins de ne pas aller trop vite : le C# ressemble à d'autres langages mais il a quand même ses spécificités ! Nous allons découvrir ensemble de nombreuses choses en apprenant à développer en C#. Il y aura bien entendu des TP pour vous faire pratiquer, afin que vous puissiez vous rendre compte de ce que vous êtes capables de faire après avoir lu plusieurs chapitres plus théoriques. Néanmoins, je veux que vous soyez actifs ! Ne vous contentez pas de lire passivement mes explications, même lorsque les chapitres sont plutôt théoriques ! Testez les codes et les manipulations au fur et à mesure. Essayez les petites idées que vous avez pour améliorer ou adapter légèrement le code. Sortez un peu des sentiers battus du tutoriel : cela vous fera pratiquer et vous permettra de découvrir rapidement si vous avez compris ou non le chapitre. Pas d'inquiétude, si jamais vous bloquez sur quoi que ce soit qui n'est pas expliqué dans ce cours, la communauté qui sillonne les forums du Site du Zéro saura vous apporter son aide précieuse.

Suivez l'ordre des chapitres

Lisez ce livre comme on lit un roman. Il a été conçu pour cela.

Contrairement à beaucoup de livres techniques où il est courant de lire en diagonale et de sauter certains chapitres, il est ici très fortement recommandé de suivre l'ordre du cours, à moins que vous ne soyez déjà un peu expérimentés.

Utilisez les codes web !

Afin de tirer parti du Site du Zéro dont ce livre est issu, celui-ci vous propose ce qu'on appelle des « codes web ». Ce sont des codes à six chiffres à saisir sur une page du Site du Zéro pour être automatiquement redirigé vers un site web sans avoir à en recopier l'adresse.

Pour utiliser les codes web, rendez-vous sur la page suivante [2] :

`http://www.siteduzero.com/codeweb.html`

Un formulaire vous invite à rentrer votre code web. Faites un premier essai avec le code ci-dessous :

▷ Tester le code web
Code web : 123456

Ces codes web ont deux intérêts :

– ils vous redirigent vers les sites web présentés tout au long du cours, vous permettant ainsi d'obtenir les logiciels dans leur toute dernière version ;
– ils vous permettent de télécharger les codes sources inclus dans ce livre, ce qui vous évitera d'avoir à recopier certains programmes un peu longs.

Ce système de redirection nous permet de tenir à jour le livre que vous avez entre les mains sans que vous ayez besoin d'acheter systématiquement chaque nouvelle édition. Si un site web change d'adresse, nous modifierons la redirection mais le code web à utiliser restera le même. Si un site web disparaît, nous vous redirigerons vers une page du Site du Zéro expliquant ce qui s'est passé et vous proposant une alternative.

En clair, c'est un moyen de nous assurer de la pérennité de cet ouvrage sans que vous ayez à faire quoi que ce soit !

Ce livre est issu du Site du Zéro

Cet ouvrage reprend le cours C# du Site du Zéro dans une édition revue et corrigée, augmentée de nouveaux chapitres plus avancés et des notes de bas de page.

Il reprend les éléments qui ont fait le succès des cours du site, à savoir leur approche progressive et pédagogique, leur ton décontracté, ainsi que les TP vous permettant de pratiquer de façon autonome.

Ce livre s'adresse donc à toute personne désireuse d'apprendre les bases de la programmation en C#, que ce soit :

– par curiosité ;
– par intérêt personnel ;
– par besoin professionnel.

Remerciements

Je souhaite remercier un certain nombre de personnes qui, de près ou de loin, ont contribué à la naissance de cet ouvrage :

– ma femme Delphine qui me soutient au quotidien et m'offre chaque jour une raison d'avancer dans la vie à ses côtés ;

2. Vous pouvez aussi utiliser le formulaire de recherche du Site du Zéro, section « Code web ».

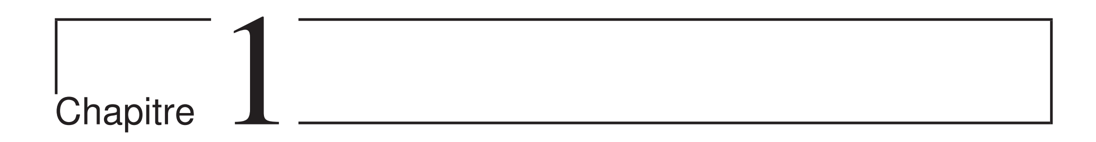
- Jérémie, mon « ami-témoin-compagnon-de-dev' », qui a bien voulu relire mes premiers essais et qui a toujours une nouvelle idée à développer ;
- Anna, Jonathan, Mathieu, Pierre et toute l'équipe de Simple IT ;
- tous les relecteurs et particulièrement Julien Patte (alias Orwell), qui m'a donné d'excellents conseils ;
- tous les lecteurs qui ont contribué à son amélioration grâce à leurs commentaires précieux et leur envie de voir le livre terminé.

Bonne lecture !

Sommaire

Première partie

Les rudiments du langage C#



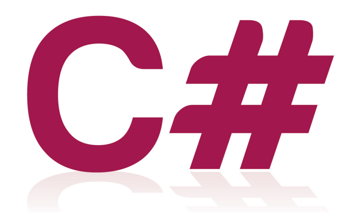
Introduction au C#

Difficulté : 

Dans ce tout premier chapitre, nous allons découvrir ce qu'est le C#, son histoire et son rapport avec le framework .NET. D'ailleurs, vous ne savez pas ce qu'est un framework ? Ce n'est pas grave, tout ceci sera expliqué !

Nous verrons dans ce chapitre ce que sont les applications informatiques et comment des langages de programmation évolués comme le C# nous permettent de réaliser de telles applications.

Et ce n'est que le début. . . alors ouvrez grands vos yeux, chaussez vos lunettes et explorons ce monde merveilleux !



Qu'est-ce que le C# ?

Le C# est un langage de programmation créé par Microsoft en 2002.



Un langage de programmation est un ensemble d'instructions, c'est-à-dire un ensemble de mots qui permettent de construire des applications informatiques.

Ces applications informatiques peuvent être de beaucoup de sortes, par exemple une application Windows, comme un logiciel de traitement de texte, une calculatrice ou encore un jeu de cartes. On les appelle également des **clients lourds**. Il est également possible de développer des applications web, comme un site d'e-commerce, un intranet, etc. Nous pourrons accéder à ces applications grâce à un navigateur internet que l'on appelle un **client léger**. Toujours grâce à un navigateur internet, nous pourrons développer des **clients riches**. Ce sont des applications qui se rapprochent d'une application Windows mais qui fonctionnent dans un navigateur. Bien d'autres types d'applications peuvent être écrites avec le C#, citons encore le développement d'applications mobiles sous Windows phone 7, de jeux ou encore de web services... Nous verrons un peu plus en détail à la fin de cet ouvrage comment réaliser de telles applications !

Le C# est un langage dont la syntaxe ressemble un peu au C++ ou au Java qui sont d'autres langages de programmation très populaires. Le C# est le langage phare de Microsoft. Il fait partie d'un ensemble plus important : il est en fait une brique de ce qu'on appelle le « **Framework .NET** ». Gardons encore un peu de suspens sur ce qu'est le framework .NET, nous découvrirons ce que c'est un peu plus loin dans ce livre.

Comment sont créées les applications informatiques ?

Une application informatique : qu'est-ce que c'est ?

Comme vous le savez, votre ordinateur exécute des applications informatiques pour effectuer des tâches. Ce sont des logiciels comme :

- un traitement de texte ;
- un navigateur internet ;
- un jeu vidéo ;
- etc.

Votre ordinateur ne peut exécuter ces applications informatiques que si elles sont écrites dans le seul langage qu'il comprend, le binaire. Techniquement, le binaire est représenté par une suite de 0 et de 1, comme vous pouvez le voir à la figure 1.1.

Il n'est bien sûr pas raisonnablement possible de réaliser une grosse application en binaire, c'est pour ça qu'il existe des langages de programmation qui permettent de simplifier l'écriture d'une application informatique.



FIGURE 1.1 – Le langage binaire est composé de 0 et de 1

Comment créer des programmes « simplement » ?

Je vais vous expliquer rapidement le principe de fonctionnement des langages dits traditionnels, comme le C et le C++, puis je vous présenterai le fonctionnement du C#. Comme le C# est plus récent, il a été possible d'améliorer son fonctionnement par rapport au C et au C++ comme nous allons le voir.

Langages traditionnels : la compilation

Avec des langages traditionnels comme le C et le C++, on écrit des instructions simplifiées, lisibles par un humain comme :

```
printf("Bonjour");
```

Ce n'est pas vraiment du français, mais c'est quand même beaucoup plus simple que le binaire et on comprend globalement avec cet exemple que l'on va afficher le mot « Bonjour ».

Bien entendu, l'ordinateur ne comprend pas ces instructions. Lui, il veut du binaire, du vrai ! Pour obtenir du binaire à partir d'un code écrit en C ou C++, on doit effectuer ce qu'on appelle une **compilation**. Le compilateur est un programme qui traduit le code source en binaire exécutable. Ce n'est pas clair ? Observez donc la figure 1.2 !

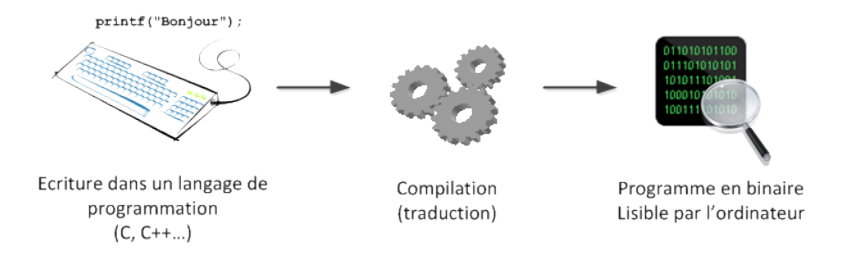


FIGURE 1.2 – Compilation traditionnelle d'un programme en binaire

Cette méthode est efficace et a fait ses preuves. De nombreuses personnes développent toujours en C et C++ aujourd'hui. Néanmoins, ces langages ont aussi un certain nombre de défauts dus à leur ancienneté. Par exemple, un programme compilé (binaire) ne fonctionne que sur la plateforme pour laquelle il a été compilé. Cela veut dire que si

vous compilez sous Windows, vous obtenez un programme qui fonctionne sous Windows uniquement (et sur un type de processeur particulier). Impossible de le faire tourner sous Mac OS X ou Linux, à moins de le recompiler sous ces systèmes d'exploitation et d'effectuer au passage quelques modifications (voir figure 1.3).

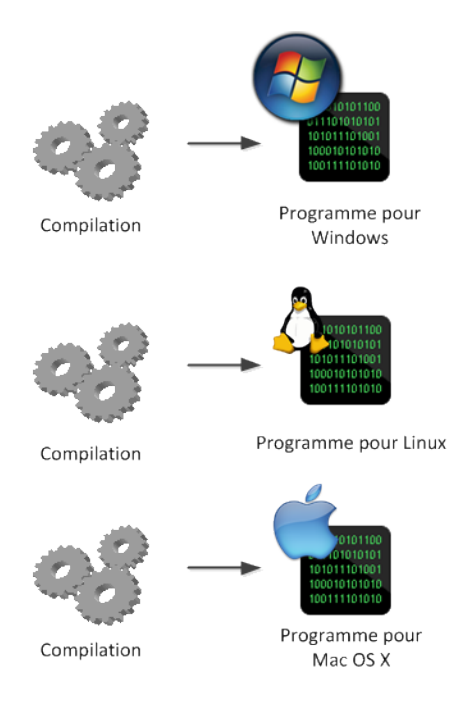


FIGURE 1.3 – Compilations multiples pour cibler toutes les plateformes

Les programmes binaires ont ce défaut : ils ne fonctionnent que pour un type de machine. Pour les développeurs qui écrivent le code, c'est assez fastidieux à gérer.

Langages récents : le code managé

Les langages récents, comme le C# et le Java, résolvent ce problème de compatibilité tout en ajoutant au langage de nombreuses fonctionnalités appréciables, ce qui permet de réaliser des programmes beaucoup plus efficacement.

La compilation en C# ne donne pas un programme binaire, contrairement au C et au C++. Le code C# est en fait transformé dans un langage intermédiaire (appelé CIL ou MSIL) que l'on peut ensuite distribuer à tout le monde. Ce code, bien sûr, n'est pas exécutable lui-même, car l'ordinateur ne comprend que le binaire.

Regardez bien la figure 1.4 pour comprendre comment cela fonctionne.

Le code en langage intermédiaire (CIL) correspond au programme que vous allez distribuer. Sous Windows, il prend l'apparence d'un `.exe` comme les programmes habituels, mais il ne contient en revanche pas de binaire.

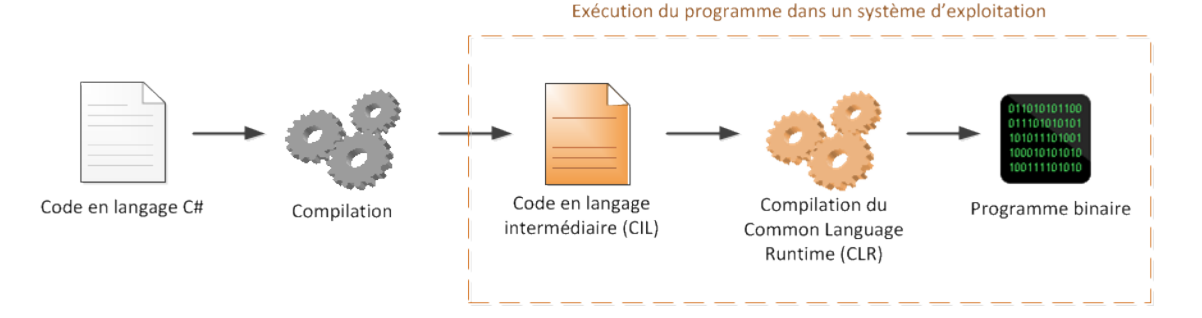


FIGURE 1.4 – Compilation C# et exécution du CIL par le CLR

Lorsqu'on exécute le programme CIL, celui-ci est lu par un autre programme (une machine à analyser les programmes, appelée CLR) qui le compile cette fois en vrai programme binaire. Cette fois, le programme peut s'exécuter, ouf !



Ça complique bien les choses quand même ! Est-ce bien utile ?

Cela offre beaucoup de souplesse au développeur. Le code en langage intermédiaire (CIL) peut être distribué à tout le monde. Il suffit d'avoir installé la machine CLR sur son ordinateur, qui peut alors lire les programmes en C# et les compiler « à la volée » en binaire. Avantage : le programme est toujours adapté à l'ordinateur sur lequel il tourne.



Le CLR vérifie aussi la sécurité du code ; ainsi en C du code mal pensé (par exemple une mauvaise utilisation des pointeurs) peut entraîner des problèmes pour votre PC, ce que vous risquez beaucoup moins avec le C#. De plus, le CLR dispose du JIT debugger qui permet de lancer Visual Studio si une erreur survient dans un programme .NET pour voir ce qui a causé cette erreur. On parle de code « managé ».

Cette complexité ralentit légèrement la vitesse d'exécution des programmes (par rapport au C ou au C++), mais la différence est aujourd'hui vraiment négligeable par rapport aux gains que cela apporte.

Donc, en théorie, il est possible d'utiliser n'importe quelle application compilée en langage intermédiaire à partir du moment où il y a une implémentation du CLR disponible. En réalité, il n'y a que sous Windows qu'il existe une implémentation complète du CLR. Il existe cependant une implémentation partielle sous Linux : Mono. Cela veut dire que si votre programme utilise des fonctionnalités qui ne sont pas couvertes par Mono, il ne fonctionnera pas.

En conclusion, dans la pratique, le .NET est totalement exploitable sous Windows et ailleurs, non.

Exécutables ou assemblages ?

J'ai dit juste au dessus que le C# était compilé en langage intermédiaire et qu'on le retrouve sous la forme d'un `.exe` comme les programmes habituels.



C'est vrai ! (Je ne mens jamais !)

Par contre, c'est un peu incomplet.

Il est possible de créer des programmes (`.exe`) qui pourront directement être exécutés par le CLR, mais il est également possible de créer des bibliothèques sous la forme d'un fichier possédant l'extension `.dll`.

On appelle ces deux formes de programme des assemblages, mais on utilise globalement toujours le mot anglais *assembly*.

– Les fichiers `.exe` sont des assemblys de processus.
– Les fichiers `.dll` sont des assemblys de bibliothèques.

Concrètement, cela signifie que le fichier `.exe` servira à lancer une application et qu'une `dll` pourra être partagée entre plusieurs applications `.exe` afin de réutiliser du code déjà écrit.

Nous verrons un peu plus loin comment ceci est possible.



Il est à noter qu'un raccourci est souvent fait avec le terme assembly. On a tendance à croire que le mot assembly sert à désigner uniquement les bibliothèques dont l'extension est `.dll`.

Qu'est-ce que le framework .NET ?

J'ai commencé à vous parler du C# qui était une brique du framework .NET. Il est temps d'en savoir un peu plus sur ce fameux framework.

Commençons par le commencement : comment cela se prononce ?

DOTTE NETTE ou encore POINT NETTE. Je vous accorde que le nom est bizarre. . . ! Surtout que le nom peut être trompeur. Avec l'omniprésence d'internet, son abréviation (net) ou encore des noms de domaines (.net), on pourrait penser que le framework .NET est un truc dédié à internet. Que nenni !

Nous allons donc préciser un peu ce qu'est le framework .NET pour éviter les ambiguïtés.



Première chose à savoir, qu'est-ce qu'un framework ?

Pour simplifier, on peut dire qu'un framework est une espèce de grosse **boîte à fonctionnalités** qui va nous permettre de réaliser des applications informatiques de toutes sortes.



En fait, c'est la combinaison de ce framework et du langage de programmation C# qui va nous permettre de réaliser ces applications informatiques.

Le framework .NET a été créé par Microsoft en 2002, en même temps que le C#, qui est principalement dédié à la réalisation d'applications fonctionnant dans des environnements Microsoft. Nous pourrons par exemple réaliser des programmes qui fonctionnent sous Windows, ou bien des sites web ou encore des applications qui fonctionnent sur téléphone mobile, etc.

Disons que la réalisation d'une application informatique, c'est un peu comme un chantier (je ne dis pas ça parce qu'il y a toujours du retard, même si c'est vrai !). Il est possible de construire différentes choses, comme une maison, une piscine, une terrasse, etc. Pour réaliser ces constructions, nous allons avoir besoin de matériaux, comme des briques, de la ferraille, etc. Certains matériaux sont communs à toutes les constructions (fer, vis,. . .) et d'autres sont spécifiques à certains domaines (pour construire une piscine, je vais avoir besoin d'un liner par exemple).

On peut voir le framework .NET comme ces matériaux ; c'est un ensemble de composants que l'on devra assembler pour réaliser notre application. Certains sont spécifiques pour la réalisation d'applications web, d'autres pour la réalisation d'applications Windows, etc.

Pour réaliser un chantier, nous allons avoir besoin d'outils afin de manipuler les matériaux. Qui envisagerait de tourner une vis avec les doigts ou de poser des parpaings sans les coller avec du mortier ? C'est la même chose pour une application informatique, pour assembler notre application, nous allons utiliser un langage de programmation : le C#.

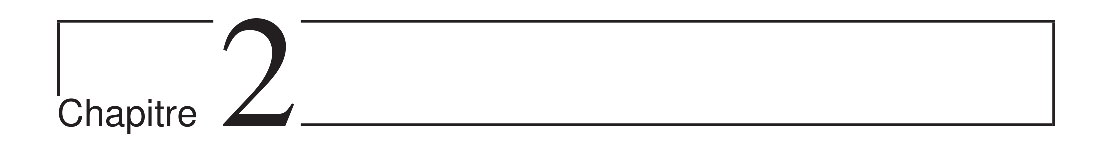
À l'heure où j'écris ces lignes, le C# est en version 4 ainsi que le framework .NET. Ce sont des versions stables et utilisées par beaucoup de personnes. Chaque version intermédiaire a apporté son lot d'évolutions. Le framework .NET et le C# sont en perpétuelle évolution, preuve de la dynamique apportée par Microsoft.

C'est tout ce qu'il y a à savoir pour l'instant ! Nous reviendrons un peu plus en détail sur le framework .NET dans les chapitres suivants. Pour l'heure, il est important de retenir que c'est grâce au langage de programmation C# et grâce aux composants du framework .NET que nous allons pouvoir développer des applications informatiques.

En résumé

- Le C# est un langage de programmation permettant d'utiliser le framework .NET. C'est le langage phare de Microsoft.
- Le framework .NET est une énorme boîte à fonctionnalités permettant la création

d'applications.
- Le C# permet de développer des applications de toutes sortes, exécutables par le CLR qui traduit le MSIL en binaire.
- Il est possible de créer des assemblys de deux sortes : des assemblys de processus exécutables par le CLR et des assemblys de bibliothèques.



Chapitre 2

Créer un projet avec Visual C# 2010 Express

Difficulté : 

Dans ce chapitre nous allons faire nos premiers pas avec le C#. Nous allons dans un premier temps installer et découvrir les outils qui nous seront nécessaires pour réaliser des applications informatiques avec le C#. Nous verrons comment démarrer avec ces outils et à la fin de ce chapitre, nous serons capables de créer un petit programme qui affiche du texte simple et nous aurons commencé à nous familiariser avec l'environnement de développement.

Il faut bien commencer par les bases, mais vous verrez comme cela peut être gratifiant d'arriver enfin à faire un petit quelque chose. Allez, c'est parti !



Que faut-il pour démarrer ?

J'espère vous avoir donné envie de démarrer l'apprentissage du C#, cependant, il nous manque encore quelque chose pour pouvoir sereinement attaquer cet apprentissage.

Bien sûr, vous allez avoir besoin d'un ordinateur, mais a priori, vous l'avez déjà ! S'il n'est pas sous Windows, mais sous Linux, vous pouvez utiliser Mono qui va permettre d'utiliser le C# sous Linux. Il est téléchargeable via le code web suivant :

▷ Télécharger Mono
Code web : 566528

Cependant, Mono n'est pas aussi complet que le C# et le framework .NET sous Windows. En l'utilisant vous risquez de passer à côté de certaines parties de ce livre.

Pour reprendre la métaphore du chantier, on peut dire qu'il va également nous manquer un chef de chantier. Il n'est pas forcément nécessaire en théorie, mais dans la pratique il se révèle indispensable pour mener à bien son chantier.

Ce chef de chantier c'est en fait l'**outil de développement**. Il va nous fournir les outils pour orchestrer nos développements.

C'est entre autres :

- **un puissant éditeur**, qui va nous servir à créer des fichiers contenant des instructions en langage C# ;
- **un compilateur**, qui va servir à transformer ces fichiers en une suite d'instructions compréhensibles par l'ordinateur, comme nous l'avons déjà vu ;
- **un environnement d'exécution**, qui va permettre de faire les actions informatiques correspondantes (afficher une phrase, réagir au clic de la souris, etc.) ; c'est le CLR dont on a déjà parlé.

Enfin, nous aurons besoin d'une **base de données**. Nous y reviendrons plus en détail ultérieurement, mais la base de données est un endroit où seront stockées les données de notre application. C'est un élément indispensable à mesure que l'application grandit.

Installer Visual C# 2010 Express

Nous avons donc besoin de notre chef de chantier, l'outil de développement. C'est un logiciel qui va nous permettre de créer des applications et qui va nous fournir les outils pour orchestrer nos développements. La gamme de Microsoft est riche en outils professionnels de qualité pour le développement, notamment grâce à **Visual Studio**.



Notez que cet outil de développement se nomme également un IDE pour *Integrated Development Environment* ce qui signifie « Environnement de développement intégré ».

Nous aurons recours au terme IDE régulièrement.

Pour apprendre et commencer à découvrir l'environnement de développement, Micro-

soft propose gratuitement Visual Studio dans sa version Express. C'est une version allégée de l'environnement de développement qui permet de faire plein de choses, mais avec des outils en moins par rapport aux versions payantes. Rassurez-vous, ces versions gratuites sont très fournies et permettent de faire tout ce dont on a besoin pour apprendre le C# et suivre ce cours.

Pour réaliser des applications d'envergure, il pourra cependant être judicieux d'investir dans l'outil complet et ainsi bénéficier de fonctionnalités complémentaires qui permettent d'améliorer, de faciliter et d'industrialiser les développements.

Pour développer en C# gratuitement et créer des applications Windows, nous allons avoir besoin de **Microsoft Visual C# 2010 Express** que vous pouvez télécharger en suivant ce code web :

▷ Visual C# 2010 Express
Code web : 389622

Pour résumer :

– Visual Studio est la version payante de l'outil de développement.
– Microsoft Visual C# 2010 Express est une version allégée et gratuite de Visual Studio, dédiée au développement en C#. Exactement ce qu'il nous faut !

Cliquez sur Visual C# 2010 Express et choisissez la langue qui vous convient. Puis cliquez sur `Téléchargez` (voir les figures 2.1 et 2.2).



FIGURE 2.1 – Page de téléchargement de Visual C# Express 2010

Une fois l'exécutable téléchargé, il ne reste plus qu'à le lancer et l'installation démarre. Cliquez sur `Suivant` pour démarrer l'installation, ainsi qu'indiqué à la figure 2.3.

Vous devez ensuite lire la licence d'utilisation du logiciel et l'accepter pour pouvoir continuer l'installation.

Une application sans données, c'est plutôt rare. C'est un peu comme un site d'e-

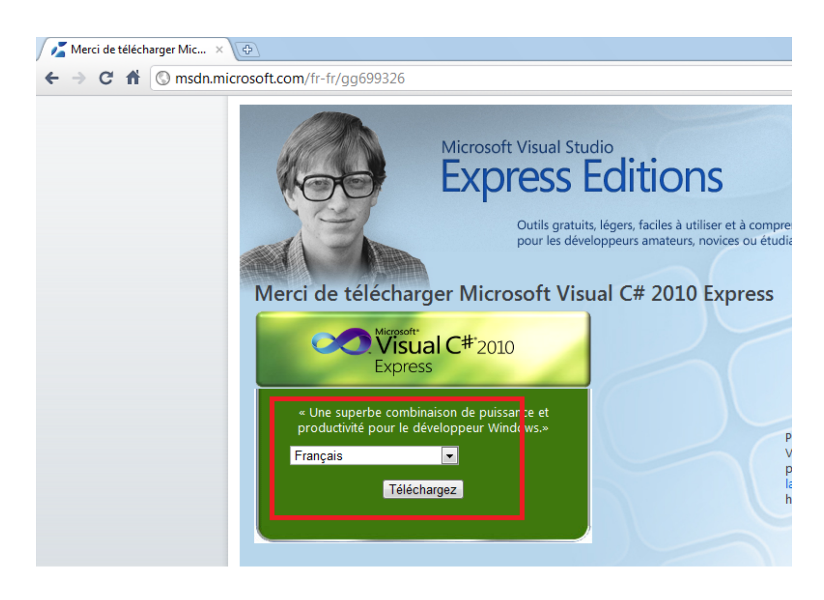


Figure 2.2 – Téléchargement de Visual C# Express 2010

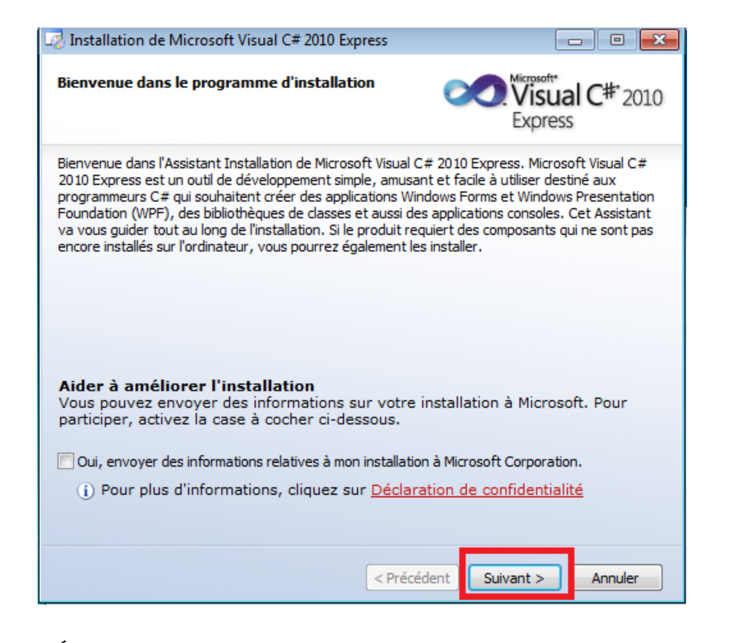


Figure 2.3 – Écran d'accueil du programme d'installation de Visual C# Express

commerce sans produits, un traitement de texte sans fichiers ou le site du zéro sans tutoriel. On risque de vite s'ennuyer !

Heureusement, le programme d'installation nous propose d'installer « Microsoft SQL Server 2008 Express Service Pack 1 ». Microsoft propose en version gratuite un serveur de base de données allégé. Il va nous permettre de créer facilement une base de données et de l'utiliser depuis nos applications en C#.



Nous l'avons déjà évoqué et nous y reviendrons plus en détail dans un chapitre ultérieur mais une base de données est un énorme endroit où sont stockées les données de notre application.

Nous avons également évoqué dans l'introduction qu'il était possible de réaliser des applications qui ressemblent à des applications Windows mais dans un navigateur, que nous avons appelé **client riche**. Silverlight va justement permettre de créer ce genre d'application.

Cochez donc tout pour installer Silverlight et Sql Server et cliquez sur `Suivant` (voir figure 2.4).

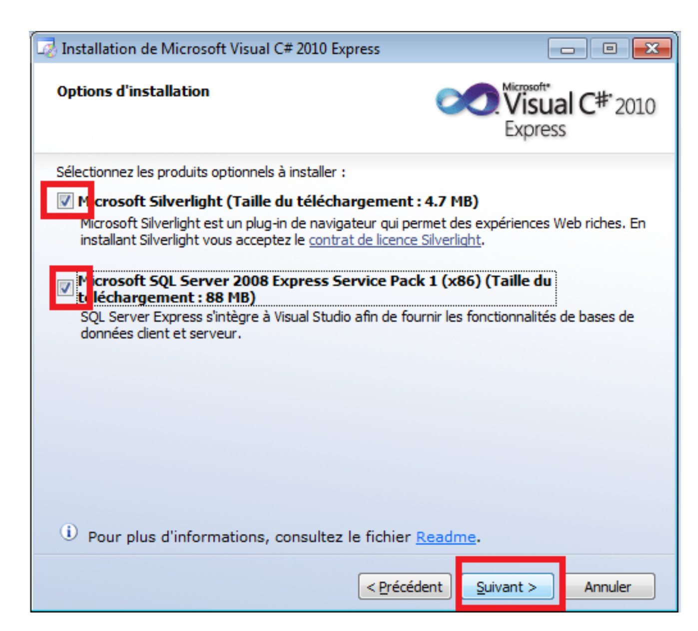


FIGURE 2.4 – Installer Silverlight et Microsoft Sql Server Express 2008

Cliquez ensuite sur `Installer` en changeant éventuellement le dossier d'installation. Une fois l'installation terminée cliquez sur `Quitter`.

À l'heure où j'écris ces lignes, il existe un service pack pour Visual Studio, le « Service pack 1 ». C'est un ensemble de corrections qui permettent d'améliorer la stabilité du logiciel. Je vous invite à télécharger et installer ce pack, disponible via le code web suivant :

▷ Service pack 1
Code web : 974305

Vous voilà avec votre copie de Visual C# Express qui va vous permettre de créer des programmes en C# gratuitement et facilement. L'installation de l'outil de développement est terminée.



Notez que, bien que gratuite, vous aurez besoin d'enregistrer votre copie de Visual C# Express avant 30 jours. C'est une opération rapide et nécessitant un compte Windows Live. Après cela, vous pourrez utiliser Visual C# Express sans retenue.

En résumé, nous avons installé un outil de développement, Visual C# 2010 dans sa version Express et une base de données, SQL Server Express 2008. Nous avons tous les outils nécessaires et nous allons pouvoir démarrer (enfin !) l'apprentissage et la pratique du C#.

Démarrer Visual C# 2010 Express

Nous allons vérifier que l'installation de Visual C# Express a bien fonctionné. Et pour ce faire, nous allons le démarrer et commencer à prendre en main ce formidable outil de développement.

Il vous semblera sûrement très complexe au début mais vous allez voir, si vous suivez ce livre pas à pas, vous allez apprendre les fonctionnalités indispensables. Elles seront illustrées par des copies d'écran vous permettant de plus facilement vous y retrouver. À force d'utiliser Visual C# Express, vous verrez que vous vous sentirez de plus en plus à l'aise et peut-être oserez-vous aller fouiller dans les menus ?

Commencez par démarrer Visual C# 2010 Express. Le logiciel s'ouvre sur une page de démarrage (voir la figure 2.5).

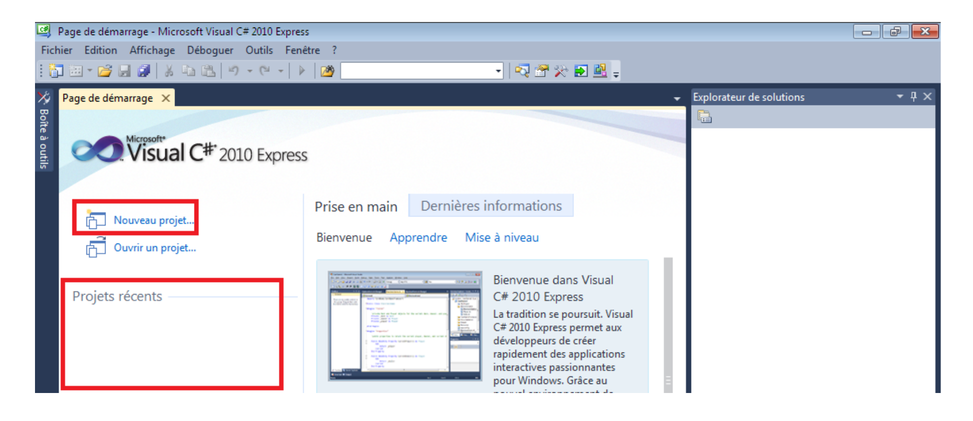


FIGURE 2.5 – Premier démarrage de Visual C# Express

Les deux zones entourées de rouge permettent respectivement de créer un nouveau projet et d'accéder aux anciens projets déjà créés. Dans ce deuxième cas, comme je viens d'installer le logiciel, la liste est vide.

Créer un projet

Commençons par créer un nouveau projet en cliquant dans la zone rouge. Cette commande est également accessible via le menu `Fichier` > `Nouveau` > `Projet`.

Un projet va contenir les éléments de ce que l'on souhaite réaliser. Cela peut être par exemple une application web, une application Windows, etc.

Le projet est aussi un container de fichiers et notamment dans notre cas de fichiers en langage C# qui vont permettre de construire ce que l'on souhaite réaliser. Le projet est en fait représenté par un fichier dont l'extension est `.csproj`. Son contenu décrit les paramètres de configuration correspondant à ce que l'on souhaite réaliser et les fichiers qui composent le projet.

Créons donc un nouveau projet. La fenêtre de création de nouveau projet s'ouvre et nous avons plusieurs possibilités de choix. Nous allons dans un premier temps aller dans `Visual C#` pour choisir de créer une `Application console`, comme indiqué à la figure 2.6.

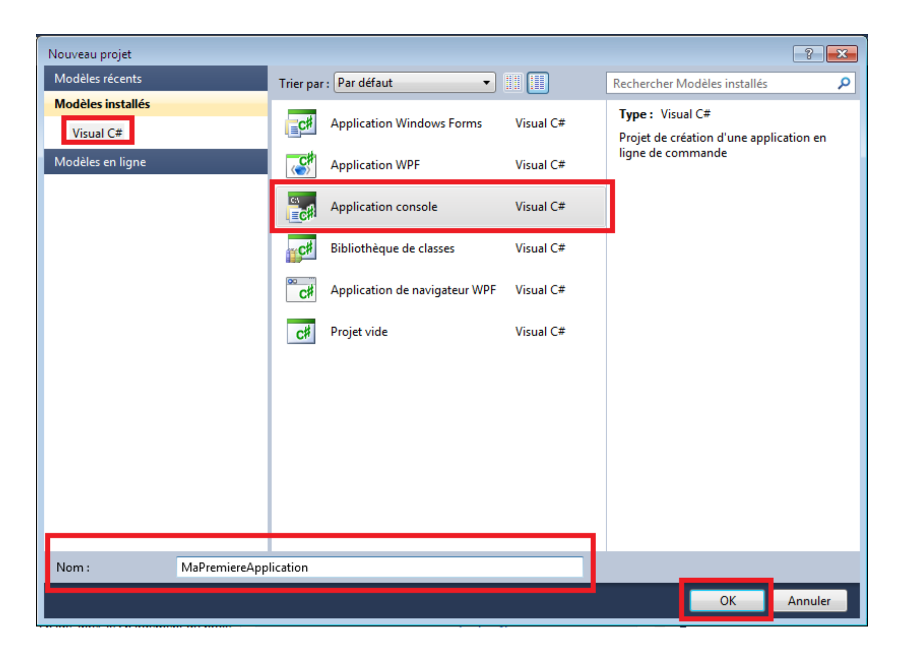


FIGURE 2.6 – Créer une application console avec Visual C# Express



À noter que si vous n'avez installé que Visual C# Express, vous aurez la même fenêtre que moi. Si vous disposez de la version payante de Visual Studio, alors la fenêtre sera certainement plus garnie. De même, il y aura plus de choses si vous avez installé d'autres outils de la gamme Express.

Ce que nous faisons ici, c'est utiliser ce qu'on appelle un « modèle »de création de projet, plus couramment appelé par son équivalent anglais *template*.

Si vous naviguez à l'intérieur des différents modèles, vous pourrez constater que Visual C# nous propose des modèles de projets plus ou moins compliqués. Ces modèles sont très utiles pour démarrer un projet car toute la configuration du projet est déjà faite. Le nombre de modèles peut être différent en fonction de votre version de Visual Studio ou du nombre de versions Express installées.

L'application console est la forme de projet pouvant produire une application exécutable la plus simple. Elle permet de réaliser un programme qui va s'exécuter dans la console noire qui ressemble à une fenêtre ms-dos, pour les dinosaures comme moi qui ont connu cette époque... À noter que les projets de type « Bibliothèque de classes » permettent de générer des assemblys de bibliothèques (`.dll`).

Dans cette console, nous allons notamment pouvoir afficher du texte simple. Ce type de projet est parfait pour démarrer l'apprentissage du C# car il y a seulement besoin de savoir comment afficher du texte pour commencer.

En bas de la fenêtre de création de projet, nous avons la possibilité de choisir un nom pour le projet, ici `ConsoleApplication1`. Changeons le nom de notre application en, par exemple, `MaPremiereApplication`, dans la zone correspondante.

Cliquons sur `OK` pour valider la création de notre projet.

Visual C# Express crée alors pour nous les fichiers composant une application console vide, qui utilise le C# comme langage et que nous avons nommée « MaPremiereApplication ».

Avant toute chose, nous allons enregistrer le projet. Allons dans le menu `Fichier` > `Enregistrer` (ou utiliser le raccourci bien connu `Ctrl` + `S`).

Visual C# Express nous ouvre la fenêtre d'enregistrement de projet (voir figure 2.7).

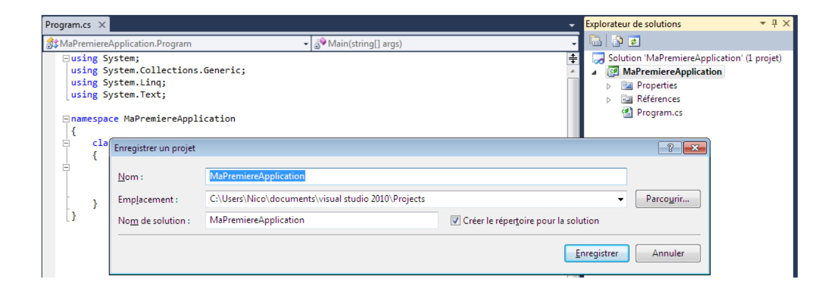


FIGURE 2.7 – Enregistrer la solution

Nous pouvons donner un nom, préciser un emplacement où nous souhaitons que les

fichiers soient enregistrés et un nom de solution. Une case pré-cochée nous propose de créer un répertoire pour la solution. C'est ce que nous allons faire ! Cliquons maintenant sur `Enregistrer`.



Pour les versions payantes de Visual Studio, le choix de l'emplacement et le nom de la solution sont à renseigner au moment où l'on crée le projet. Une différence subtile !

Analyse de l'environnement de développement et du code généré

Si l'on se rend dans l'emplacement renseigné (ici `c:\users\nico\documents\visual studio 2010\Projects`), nous pouvons constater que Visual C# Express a créé un répertoire `MaPremiereApplication` : c'est le fameux répertoire pour la solution qu'il nous a proposé de créer.

Dans ce répertoire, nous remarquons notamment un fichier `MaPremiereApplication.sln`.

C'est ce qu'on appelle **le fichier de solution** ; il s'agit juste d'un container de projets qui va nous permettre de visualiser nos projets dans Visual C# Express.

En l'occurrence, nous n'avons pour l'instant qu'un projet dans la solution : l'application `MaPremiereApplication`, que nous pouvons retrouver dans le sous-répertoire `MaPremiereApplication` et qui contient notamment le fichier de projet qui s'intitule : `MaPremiereApplication.csproj`.



Le fichier décrivant un projet écrit en C# est préfixé par `csproj`.

Il y a encore un fichier digne d'intérêt dans ce répertoire, il s'agit du fichier `Program.cs`. Les fichiers dont l'extension est `.cs` contiennent du code C#, c'est dans ce fichier que nous allons commencer à taper nos premières lignes de code...



L'ensemble des fichiers contenant des instructions écrites dans un langage de programmation est appelé le « code source ». Par extension, le « code » correspond à des instructions écrites dans un langage de programmation.

Retournons dans l'interface de Visual C# Express, et détaillons un peu les différentes zones ! Pour suivre mes explications, reportez-vous à la figure 2.8.

- La zone verte **numéro 1** contient les différents fichiers ouverts sous la forme d'un onglet. On voit que, par défaut, Visual C# nous a créé et ouvert le fichier `Program.cs`.
- Dans la zone rouge **numéro 2**, c'est l'éditeur de code. Il affiche le contenu du fichier ouvert. Nous voyons des mots que nous ne comprenons pas encore. C'est du code qui a été automatiquement généré par Visual C#. Nous pouvons observer que les mots

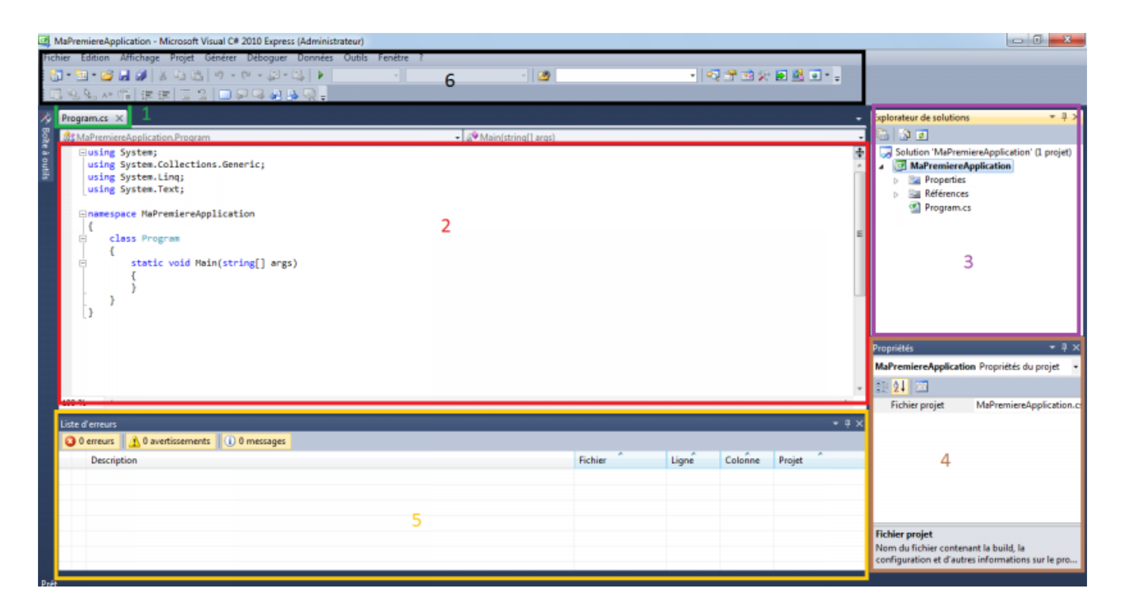


Figure 2.8 – L'IDE de Visual C# Express

sont de différentes couleurs. En effet, l'éditeur Visual C# Express possède ce qu'on appelle une **coloration syntaxique**, c'est-à-dire que certains mots-clés sont colorés d'une couleur différente en fonction de leur signification ou de leur contexte afin de nous permettre de nous y retrouver plus facilement. Comme dans ce livre !

– La zone **numéro 3** en violet est l'explorateur de solutions. C'est ici que l'on voit le contenu de la solution sur laquelle nous travaillons en ce moment. En l'occurrence, il s'agit de la solution `MaPremiereApplication` qui contient un unique projet `MaPremiereApplication`. Ce projet contient plusieurs sous éléments :

  1. `Properties` : contient des propriétés de l'application, on ne s'en occupe pas pour l'instant ;
  2. `Références` : contient les références de l'application, on ne s'en occupe pas non plus pour l'instant ;
  3. `Program.cs` est le fichier qui a été généré par Visual C# et qui contient le code C#. Il nous intéresse fortement !

– La zone **numéro 4** en brun est la zone contenant les propriétés de ce sur quoi nous travaillons en ce moment. Ici, nous avons le curseur positionné sur le projet, il n'y a pas beaucoup d'informations excepté le nom du fichier de projet. Nous aurons l'occasion de revenir sur cette fenêtre plus tard.

– La zone **numéro 5** en jaune n'est pas affichée au premier lancement, elle contient la liste des erreurs, des avertissements et des messages de notre application. Nous verrons comment l'afficher un peu plus bas.

– La zone **numéro 6** en noir est la barre d'outils, elle possède plusieurs boutons que nous pourrons utiliser, notamment pour exécuter notre application.

Écrire du texte dans notre application

Allons donc dans la zone 2 réservée à l'édition de notre fichier `Program.cs` qui est le fichier contenant le code C# de notre application.

Les mots présents dans cette zone sont ce qu'on appelle des instructions de langage. Elles vont nous permettre d'écrire notre programme. Nous reviendrons plus loin sur ce que veulent dire les instructions qui ont été générées par Visual C#. Pour l'instant, rajoutons simplement l'instruction suivante après l'accolade ouvrante :

```
Console.WriteLine("Hello World !!");
```

de manière à avoir :

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World !!");
}
```

Nous venons d'écrire une instruction qui va afficher la phrase « Hello World !! ». Pour l'instant vous avez juste besoin de savoir ça. Nous étudierons ultérieurement à quoi cela correspond exactement.

L'exécution du projet

Ça y est ! Nous avons écrit notre premier code qui affiche un message très populaire. Mais pour le moment, ça ne fait rien... On veut voir ce que ça donne !

Comme je vous comprends !

La première chose à faire est de transformer le langage C# que nous venons d'écrire en programme exécutable. Cette phase s'appelle la « génération de la solution » sous Visual C#. On l'appelle souvent la « **compilation** » ou en anglais le ***build*** .

Allez dans le menu `Déboguer` et cliquez sur `Générer la solution` (voir figure 2.9).

Visual C# lance alors la génération de la solution et on voit dans la barre des tâches en bas à gauche qu'il travaille jusqu'à nous indiquer que la génération a réussi, ainsi que vous pouvez le voir sur la figure 2.10.

Si nous allons dans le répertoire contenant la solution, nous pouvons voir dans le répertoire `MaPremiereApplication\MaPremiereApplication\bin\Release` qu'il y a deux fichiers :

- `MaPremiereApplication.exe`
- `MaPremiereApplication.pdb`

Le premier est le fichier exécutable, possédant l'extension `.exe`, qui est le résultat du processus de génération. Il s'agit bien de notre application.

Le second est un fichier particulier qu'il n'est pas utile de connaître pour l'instant, nous allons l'ignorer !

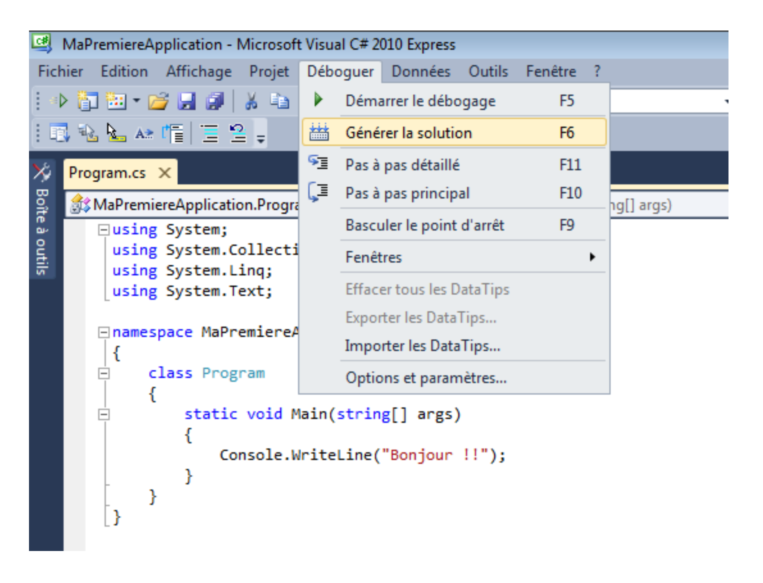


Figure 2.9 – Générer la solution

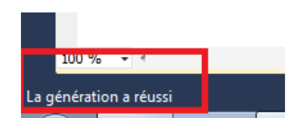


Figure 2.10 – Génération réussie !

Exécutons notre application en lançant le fichier exécutable depuis l'explorateur de fichiers. Déception, nous voyons à peine un truc noirâtre qui s'affiche et qui se referme immédiatement. Que s'est-il passé ? En fait, l'application s'est lancée, a affiché notre message et s'est terminée immédiatement. Et tout ça un brin trop rapidement... ça ne va pas être pratique tout ça. Heureusement, Visual C# Express arrive à la rescousse. Retournons dans notre IDE préféré. Nous allons ajouter un bouton dans la barre d'outils. J'avoue ne pas comprendre pourquoi ce bouton est manquant dans l'installation par défaut. Nous allons remédier à ce problème en cliquant sur la petite flèche qui est à côté de la barre d'outils, tout à droite, et qui nous ouvre le menu déroulant permettant d'ajouter ou supprimer des boutons. Cliquez sur `Personnaliser` (voir figure 2.11).

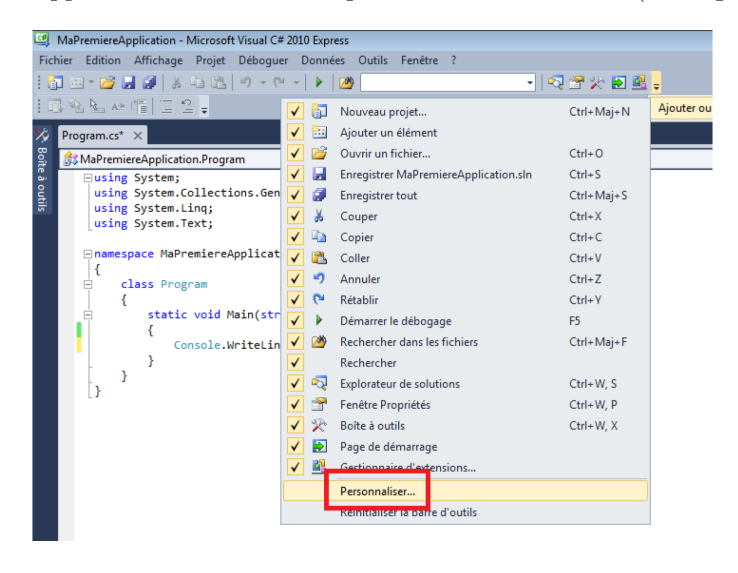


FIGURE 2.11 – Personnalisation de la barre d'outils

Cliquez sur `Ajouter une commande`, comme indiqué à la figure 2.12.

Allez dans la catégorie `Déboguer` et choisissez `Exécuter sans débogage` puis cliquez sur `OK` (voir figure 2.13).

Enfin, fermez la fenêtre. Comme vous pouvez l'observer sur la figure 2.14, vous avez désormais un nouveau bouton dans la barre d'outils !

Si vous avez la flemme d'ajouter ce bouton, vous pouvez aussi utiliser le raccourci `Ctrl` + `F5` ou bien cliquer sur ce nouveau bouton pour exécuter l'application depuis Visual C#. La console s'ouvre enfin, nous délivrant le message tant attendu (voir figure 2.15).

Le message est désormais visible car Visual C# nous demande d'appuyer sur une touche pour que l'application se termine, ce qui nous laisse donc le temps d'apprécier l'exécution de notre superbe programme.

La console noire que nous voyons sur la figure 2.15 est le résultat de l'exécution de notre programme. Si vous n'avez pas créé ce programme en même temps que moi, vous pouvez voir à quoi ressemble cette console.

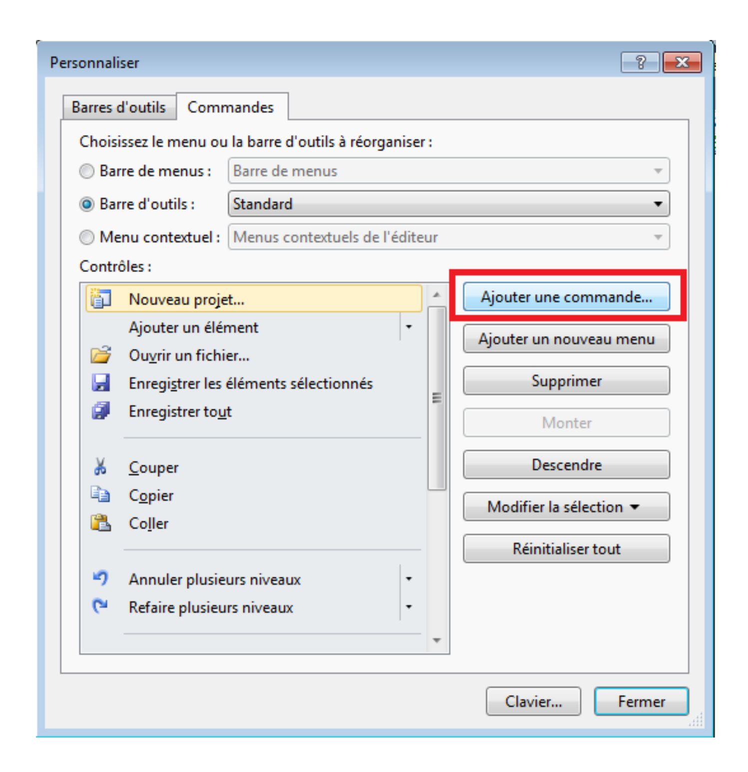


Figure 2.12 – Ajouter une commande à la barre d'outils

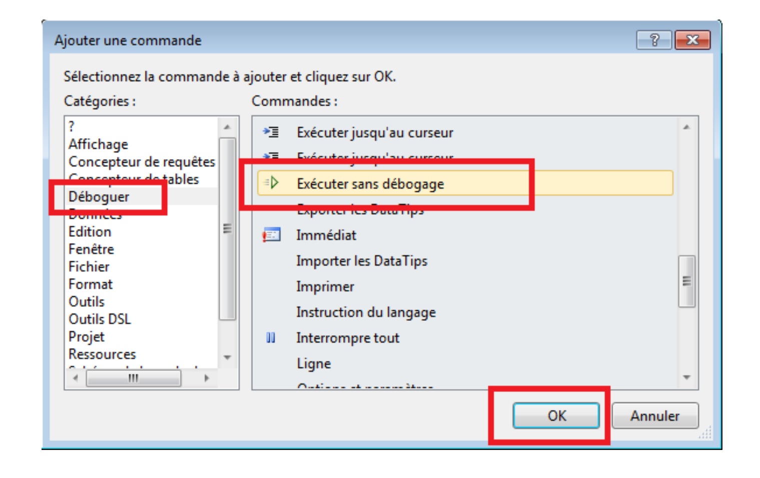


Figure 2.13 – Ajouter la commande d'exécution sans débogage

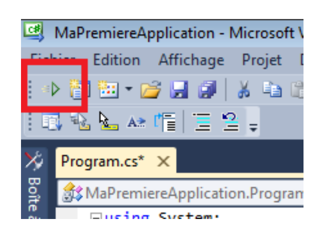


FIGURE 2.14 – Exécuter le projet sans débogage

```
C:\Windows\system32\cmd.exe
Hello World !!
Appuyez sur une touche pour continuer...
```

FIGURE 2.15 – Notre première application C# dans la console

Dans la suite du livre, nous utiliserons la syntaxe suivante pour remplacer l'image de console du dessus :

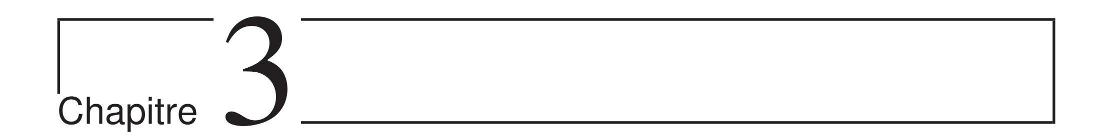
```
Hello World !!
```

Waouh, ça y est, notre première application en C# !!!

Je suis fier de nous, mais nous n'allons pas en rester là, nous sommes désormais fin parés pour apprendre le C# !

En résumé

- Visual C# Express est l'outil de développement gratuit de Microsoft permettant de démarrer avec le C#.
- Visual Studio est l'outil de développement payant de Microsoft permettant d'être efficace dans le développement d'applications .NET.
- Microsoft SQL Server Express est le moteur de base de données utilisable facilement avec Visual C# Express.
- L'environnement de développement nous permet de créer du code C# qui sera contenu dans des projets, qui peuvent être réunis dans une solution.



Chapitre 3

La syntaxe générale du C#

Difficulté : 

Nous allons aborder ici la syntaxe générale du C# dans le cadre d'une application console. Nous allons utiliser très souvent l'instruction `Console.WriteLine("...");` que nous avons vue au chapitre précédent et qui est dédiée à l'affichage sur la console. C'est une instruction qui va s'avérer très pratique pour notre apprentissage car nous pourrons avoir une représentation visuelle de ce que nous allons apprendre.

Préparez-vous, nous plongeons petit à petit dans l'univers du C#. Dans ce chapitre, nous allons nous attaquer à la syntaxe générale du C# et nous serons capables de reconnaître les lignes de code et de quoi elles se composent.



Écrire une ligne de code

Les lignes de code écrites avec le langage de développement C# doivent s'écrire dans des fichiers dont l'extension est `.cs`. Nous avons vu dans le chapitre précédent que nous avons écrit dans le fichier `Program.cs` qui est le fichier qui a été généré par Visual C# lors de la création du projet. Nous y avons notamment ajouté une instruction permettant d'afficher du texte. Les lignes de code C# se lisent et s'écrivent de haut en bas et de gauche à droite, comme un livre normal. Aussi, une instruction écrite avant une autre sera en général exécutée avant celle-ci.



Attention, chaque ligne de code doit être correcte d'un point de vue syntaxique sinon le compilateur ne saura pas la traduire en langage exécutable.

Par exemple, si à la fin de mon instruction, je retire le point-virgule ou si j'orthographie mal le mot `WriteLine`, Visual C# Express me signalera qu'il y a un problème, comme vous pouvez le voir à la figure 3.1.

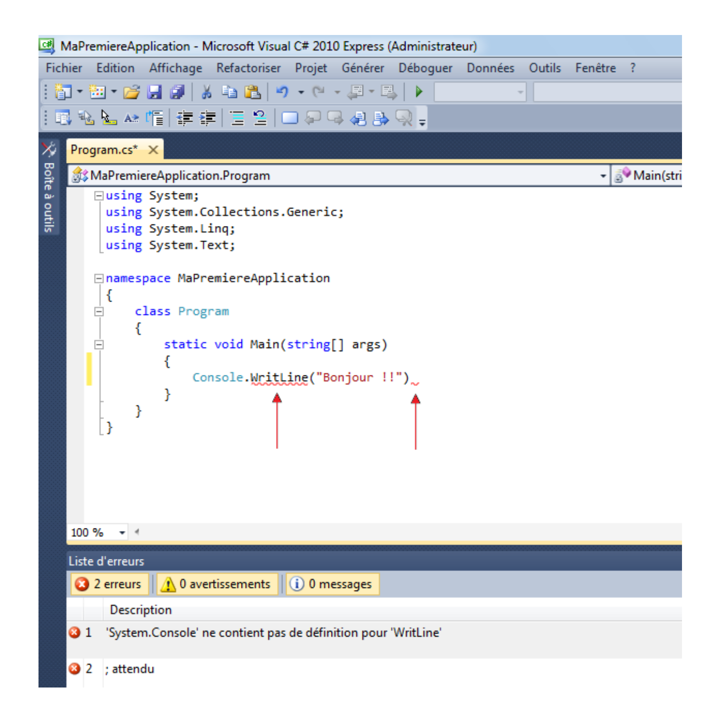


FIGURE 3.1 – Mise en valeur des erreurs de compilation

Visual C# Express met en valeur un manque au niveau de la fin de l'instruction et il me souligne également le mot « `WritLine` ». Dans la fenêtre du bas, il m'indique qu'il y a deux erreurs et me donne des précisions sur celles-ci avec éventuellement des pistes pour résoudre ces erreurs.

Si je tente de lancer mon application (avec le raccourci Ctrl + F5), Visual C# Express va tenter de compiler et d'exécuter l'application. Ceci n'étant pas possible, il m'affichera un message indiquant qu'il y a des erreurs (voir figure 3.2).

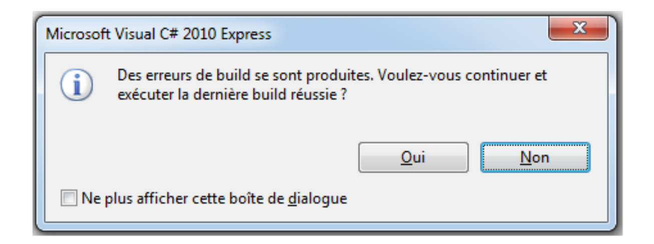


FIGURE 3.2 – Compilation impossible suite à des erreurs

Ce sont des erreurs de compilation qu'il va falloir résoudre si l'on souhaite que l'application console puisse s'exécuter.

Nous allons voir dans les chapitres suivants comment écrire correctement des instructions en C#. Mais il est important de noter à l'heure actuelle que le C# est **sensible à la casse**, ce qui veut dire que les majuscules comptent ! Ainsi le mot `WriteLine` et le mot `WriTEline` sont deux mots bien distincts et peuvent potentiellement représenter deux instructions différentes. Ici, le deuxième mot est incorrect car il n'existe pas.



Rappelez-vous bien que la casse est déterminante pour que l'application puisse compiler.

Le caractère de terminaison de ligne

En général, une instruction en code C# s'écrit sur une ligne et se termine par un **point-virgule**. Ainsi, l'instruction que nous avons vue plus haut :

```
Console.WriteLine("Hello World !!");
```

se termine au niveau du point-virgule. Il aurait été possible de remplacer le code écrit :

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World !!");
    }
}
```

par :

```
class Program {static void Main(string[] args) {Console.
    WriteLine("Hello World !!");}}
```

ou encore :

```
class Program
{
static void Main(string[] args)
    {
        Console


.WriteLine("Hello World !!"

            );
    }
}
```

En général, pour que le code soit le plus lisible possible, on écrit une instruction par ligne et on indente le code de façon à ce que les blocs soient lisibles.



Un bloc de code est délimité par des accolades : { et }. Nous y reviendrons plus tard.



Indenter signifie que chaque ligne de code qui fait partie d'un même bloc de code commence avec le même retrait sur l'éditeur. Ce sont soit des tabulations, soit des espaces qui permettent de faire ce retrait.

Visual C# Express nous aide pour faire correctement cette indentation quand nous écrivons du code. Il peut également remettre toute la page en forme avec la combinaison de touche : `Ctrl` + `K` + `Ctrl` + `D`.

Décortiquons à présent cette ligne de code :

```
Console.WriteLine("Hello World !!");
```

Pour simplifier, nous dirons que nous appelons la méthode `WriteLine` qui permet d'écrire une chaîne de caractères sur la console.



Une méthode représente une fonctionnalité, écrite avec du code, qui est utilisable par d'autres bouts de code (par exemple, calculer la racine carrée d'un nombre ou afficher du texte. . .).

L'instruction "Hello World !!" représente une chaîne de caractères et est passée en paramètre de la méthode `Console.WriteLine` à l'aide des parenthèses. La chaîne de caractères est délimitée par les guillemets. Enfin, le point-virgule permet d'indiquer que l'instruction est terminée et qu'on peut enchaîner sur la suivante.

Certains points ne sont peut-être pas encore tout à fait clairs, comme ce qu'est vraiment une méthode, ou comment utiliser des chaînes de caractères, mais ne vous inquiétez pas, nous allons y revenir plus en détail dans les chapitres suivants et découvrir au fur

et à mesure les arcanes du C#.

Les commentaires

Pour faciliter la compréhension du code ou pour se rappeler un point précis, il est possible de mettre des commentaires dans son code. Les commentaires sont ignorés par le compilateur et n'ont qu'une valeur informative pour le développeur. Dans un fichier de code C# (`.cs`), on peut écrire des commentaires de deux façons différentes :

- en commençant son commentaire par /* et en le terminant par */ ce qui permet d'écrire un commentaire sur plusieurs lignes ;
- en utilisant // et tout ce qui se trouve après sur la même ligne est alors un commentaire.

Visual C# Express colore les commentaires en vert pour faciliter leur identification.

```
/* permet d'afficher du texte
   sur la console */
Console.WriteLine("Hello World !!"); // ne pas oublier le point
    virgule
```



À noter qu'on peut commenter plusieurs lignes de code avec le raccourci clavier Ctrl + K + Ctrl + C et décommenter plusieurs lignes de code avec le raccourci clavier Ctrl + K + Ctrl + U.

La complétion automatique

Visual C# Express est un formidable outil qui nous facilite à tout moment la tâche, notamment grâce à la **complétion automatique**. La complétion automatique est le fait de proposer de compléter automatiquement ce que nous sommes en train d'écrire en se basant sur ce que nous avons le droit de faire. Par exemple, si vous avez cherché à écrire l'instruction :

```
Console.WriteLine("Hello World !!");
```

vous avez pu constater que lors de l'appui sur la touche C, Visual C# Express nous affiche une fenêtre avec tout ce qui commence par « C », ainsi que vous pouvez l'observer à la figure 3.3.

Au fur et à mesure de la saisie, il affine les propositions pour se positionner sur la plus pertinente. Il est possible de valider la proposition en appuyant sur la touche Entrée. Non seulement cela nous économise des appuis de touches, paresseux comme nous sommes, mais cela nous permet également de vérifier la syntaxe de ce que nous écrivons et d'obtenir également une mini-aide sur ce que nous essayons d'utiliser (voir figure 3.4).

```
using System.Text;

namespace MaPremiereApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            C
        }
    }
}
```


FIGURE 3.3 – La complétion automatique facilite l'écriture du code

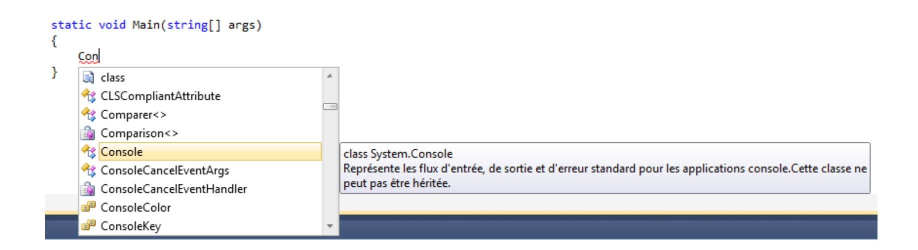


FIGURE 3.4 – La complétion automatique s'affine au fur et à mesure de la saisie

Ainsi, fini les fautes de frappe qui entraînent une erreur de compilation ou les listes de mots-clés dont il faut absolument retenir l'orthographe !

De la même façon, une fois que vous avez fini de saisir « Console » vous allez saisir le point « . » et Visual C# Express va nous proposer toute une série d'instructions en rapport avec le début de l'instruction (voir figure 3.5).

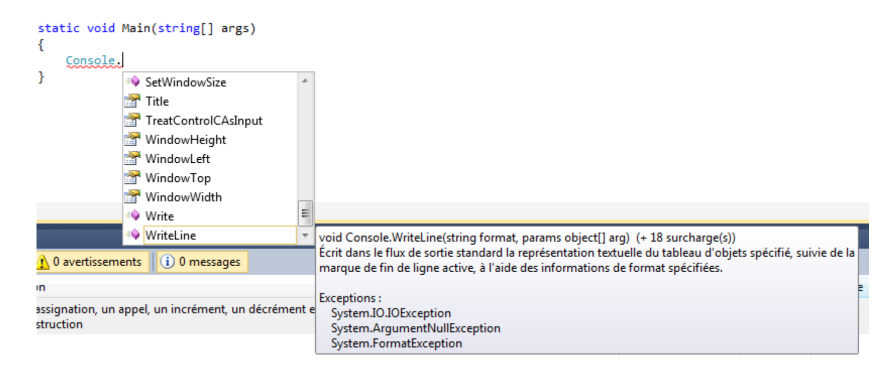
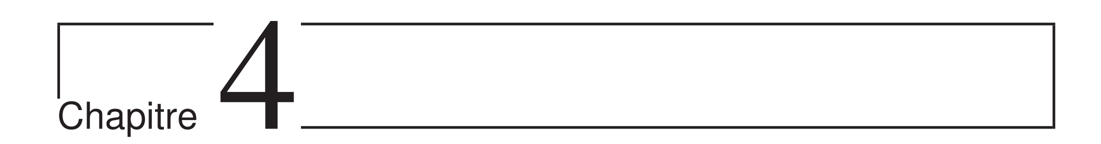


FIGURE 3.5 – La complétion automatique

Nous pourrons ainsi facilement finir de saisir `WriteLine` et ceci sans erreur d'écriture, ni problème de majuscule.

En résumé

- Le code C# est composé d'une suite d'instructions qui se terminent par un point-virgule.
- La syntaxe d'un code C# doit être correcte sinon nous aurons des erreurs de compilation.
- Il est possible de commenter son code grâce aux caractères « // », « /* » et « */ ».
- Visual C# Express dispose d'un outil puissant qui permet d'aider à compléter ses instructions : la complétion automatique.



Chapitre 4

Les variables

Difficulté : 

Dans ce chapitre nous allons apprendre ce que sont les variables et comment ces éléments indispensables vont nous rendre bien des services pour traiter de l'information susceptible de changer dans nos programmes informatiques. Nous continuerons en découvrant les différents types de variables et nous ferons nos premières manipulations avec elles.

Soyez attentifs à ce chapitre, il est vraiment fondamental de bien comprendre à quoi servent les variables lors du développement d'une application informatique.



Qu'est-ce qu'une variable ?

Comme tous les langages de programmation, le C# va pouvoir conserver des données grâce à des variables. Ce sont en fait des **blocs de mémoire** qui vont contenir des nombres, des chaînes de caractères, des dates ou plein d'autres choses. Les variables vont nous permettre d'effectuer des calculs mathématiques, d'enregistrer par exemple pour un site, l'âge du visiteur, de comparer des valeurs, etc. On peut les comparer à des petits classeurs possédant une étiquette. On va pouvoir mettre des choses dans ces classeurs, par exemple, je mets 30 dans le classeur étiqueté « âge de Nicolas » et 20 dans le classeur « âge de Jérémie ». Si je veux connaître l'âge de Nicolas, je n'ai qu'à regarder dans ce classeur pour obtenir 30. Je peux également remplacer ce qu'il y a dans mon classeur par autre chose, par exemple changer 30 en 25. Je ne peux par contre pas mettre deux choses dans mon classeur, il n'a qu'un seul emplacement.

Une variable est représentée par son **nom**, caractérisée par son **type** et contient une **valeur**.



Le type correspond à ce que la variable représente : un entier, une chaîne de caractères, une date, etc.

Par exemple, l'âge d'une personne pourrait être stocké sous la forme d'un entier et accessible par la variable `age`, ce qui s'écrit en C# :

```
int age;
```

On appelle ceci « la déclaration de la variable `age` ».

Le mot-clé `int` permet d'indiquer au compilateur que la variable `age` est un entier numérique. `int` correspond au début d'*integer* qui veut dire « entier » en anglais.

Ici, la variable `age` n'a pas été initialisée, elle ne pourra pas être utilisée car le compilateur ne sait pas quelle valeur il y a dans la variable `age`.

Pour l'initialiser (on parle également « d'affecter une valeur ») on utilisera l'opérateur égal (« = »).

```
int age;
age = 30;
```

Notre variable `age` possède désormais l'entier numérique « 30 » comme valeur.

L'initialisation d'une variable peut également se faire au moment de sa déclaration. Ainsi, on pourra remplacer le code précédent par :

```
int age = 30;
```

Pour déclarer une variable en C#, on commence toujours par indiquer son type (`int`, ici un entier) et son nom (ici, `age`). Il faudra impérativement affecter une valeur à cette variable avec l'opérateur « = », soit sur la même instruction que la déclaration, soit un peu plus loin dans le code, mais dans tous les cas, avant l'utilisation de cette variable.

Nous pouvons à tout moment demander la valeur contenue dans la variable `age`, par exemple :

```
int age = 30;
Console.WriteLine(age); // affiche 30
```

Il est possible de modifier la valeur de la variable à n'importe quel moment grâce à l'emploi de l'opérateur « = » que nous avons déjà vu :

```
int age = 30;
Console.WriteLine(age); // affiche 30
age = 20;
Console.WriteLine(age); // affiche 20
```

Vous pouvez nommer vos variables à peu près n'importe comment, à quelques détails près. Les noms de variables ne peuvent pas avoir le même nom qu'un type. Il sera alors impossible d'appeler une variable `int`. Il est également impossible d'utiliser des caractères spéciaux, comme des espaces ou des caractères de ponctuation. De même, on ne pourra pas nommer une variable en commençant par des chiffres.



Il est par contre possible d'utiliser des accents dans les noms de variables, cependant ceci n'est pas recommandé et ne fait pas partie des bonnes pratiques de développement. En effet, il est souvent recommandé de nommer ses variables en anglais (langue qui ne contient pas d'accents). Vous aurez noté que, volontairement, je ne le fais pas dans ce livre afin de ne pas ajouter une contrainte supplémentaire lors de la lecture du code. Mais libre à vous de le faire !

En général, une variable commence par une minuscule et, si son nom représente plusieurs mots, on démarrera un nouveau mot par une majuscule. Par exemple :

```
int ageDuVisiteur;
```

C'est ce qu'on appelle le **camel case**.



Suivant le principe de sensibilité à la casse, il faut faire attention car `ageduvisiteur` et `ageDuVisiteur` seront deux variables différentes :

```
int ageduvisiteur = 30;
int ageDuVisiteur = 20;
Console.WriteLine(ageduvisiteur); // affiche 30
Console.WriteLine(ageDuVisiteur); // affiche 20
```



À noter un détail qui peut paraître évident, mais toutes les variables sont réinitialisées à chaque nouvelle exécution du programme. Dès qu'on démarre le programme, les classeurs sont vidés, comme si on emménageait dans de nouveaux locaux à chaque fois. Il est donc impossible de faire persister une information entre deux exécutions du programme en utilisant des variables. Pour ceci, on utilisera d'autres solutions, comme enregistrer des valeurs dans un fichier ou dans une base de données. Nous y reviendrons ultérieurement.

Les différents types de variables

Nous avons vu juste au-dessus que la variable `age` pouvait être un entier numérique grâce au mot clé `int`. Le framework .NET dispose de beaucoup de types permettant de représenter beaucoup de choses différentes.

Par exemple, nous pouvons stocker une chaîne de caractères grâce au type `string`.

```
string prenom = "nicolas";
```

ou encore un décimal avec :

```
decimal soldeCompteBancaire = 100;
```

ou encore un boolean (qui représente une valeur vraie ou fausse) avec

```
bool estVrai = true;
```



Il est important de stocker des données dans des variables ayant le bon type.

On ne peut par exemple pas stocker le prénom « Nicolas » dans un entier.

Les principaux types de base du framework .NET sont :

Vous verrez plus loin qu'il existe encore d'autres types dans le framework .NET et qu'on peut également construire les siens.

Affectations, opérations, concaténation

Il est possible d'effectuer des opérations sur les variables et entre les variables. Nous avons déjà vu comment affecter une valeur à une variable grâce à l'opérateur « = ».

```
int age = 30;
string prenom = "nicolas";
```

Type	Description
`byte`	Entier de 0 à 255
`short`	Entier de -32768 à 32767
`int`	Entier de -2147483648 à 2147483647
`long`	Entier de -9223372036854775808 à 9223372036854775807
`float`	Nombre simple précision de -3,402823e38 à 3,402823e38
`double`	Nombre double précision de -1,79769313486232e308 à 1,79769313486232e308
`decimal`	Nombre décimal convenant particulièrement aux calculs financiers (en raison de ses nombres significatifs après la virgule)
`char`	Représente un caractère
`string`	Une chaîne de caractère
`bool`	Une valeur booléenne (vrai ou faux)



Note : dans ce paragraphe, je vais vous donner plusieurs exemples d'affectations. Ces affectations seront faites sur la même instruction que la déclaration pour des raisons de concision. Mais ces exemples sont évidemment fonctionnels pour des affectations qui se situent à un endroit différent de la déclaration.

En plus de la simple affectation, nous pouvons également faire des opérations, par exemple :

```
int resultat = 2 * 3;
```

ou encore

```
int age1 = 20;
int age2 = 30;
int moyenne = (age1 + age2) / 2;
```

Les opérateurs « + », « * », « / » ou encore « - » (que nous n'avons pas encore utilisé) servent bien évidemment à faire les opérations mathématiques qui leur correspondent, à savoir respectivement l'addition, la multiplication, la division et la soustraction. Vous aurez donc sûrement deviné que la variable `resultat` contient 6 et que la moyenne vaut 25.

Il est à noter que les variables contiennent une valeur qui ne peut évoluer qu'en affectant une nouvelle valeur à cette variable. Ainsi, si j'ai le code suivant :

```
int age1 = 20;
int age2 = 30;
int moyenne = (age1 + age2) / 2;
age2 = 40;
```

la variable `moyenne` vaudra toujours 25 même si j'ai changé la valeur de la variable `age2`. En effet, lors du calcul de la moyenne, j'ai rangé dans mon classeur la valeur 25 grâce à l'opérateur d'affectation « = » et j'ai refermé mon classeur. Le fait de changer la valeur du classeur `age2` n'influence en rien le classeur `moyenne` dans la mesure où il

est fermé. Pour le modifier, il faudrait réexécuter l'opération d'affectation de la variable `moyenne`, en écrivant à nouveau l'instruction de calcul, c'est-à-dire :

```
int age1 = 20;
int age2 = 30;
int moyenne = (age1 + age2) / 2;
age2 = 40;
moyenne = (age1 + age2) / 2;
```

L'opérateur + peut également servir à concaténer des chaînes de caractères, par exemple :

```
string codePostal = "33000";
string ville = "Bordeaux";
string adresse = codePostal + " " + ville;
Console.WriteLine(adresse); // affiche : 33000 Bordeaux
```

D'autres opérateurs particuliers existent que nous ne trouvons pas dans les cours de mathématiques. Par exemple, l'opérateur ++ qui permet de réaliser une incrémentation de 1, ou l'opérateur `--` qui permet de faire une décrémentation de 1. De même, les opérateurs que nous avons déjà vus peuvent se cumuler à l'opérateur = pour simplifier une opération qui prend une variable comme opérande et cette même variable comme résultat. Par exemple :

```
int age = 20;
age = age + 10; // age contient 30 (addition)
age = age++; // age contient 31 (incrémentation de 1)
age = age--; // age contient 30 (décrémentation de 1)
age += 10; // équivalent à age = age + 10 (age contient 40)
age /= 2; // équivalent à age = age / 2 => (age contient 20)
```

Comme nous avons pu le voir dans nos cours de mathématiques, il est possible de grouper des opérations avec des parenthèses pour agir sur leurs priorités.

Ainsi, l'instruction précédemment vue :

```
int moyenne = (age1 + age2) / 2;
```

effectue bien la somme des deux âges avant de les diviser par 2, car les parenthèses sont prioritaires.

Cependant, l'instruction suivante :

```
int moyenne = age1 + age2 / 2;
```

aurait commencé par diviser `age2` par 2 et aurait ajouté `age1`, ce qui n'aurait plus rien à voir avec une moyenne. En effet, la division est prioritaire par rapport à l'addition.



Attention, la division ici est un peu particulière.

Prenons cet exemple :

```
int moyenne = 5 / 2;
Console.WriteLine(moyenne);
```

Si nous l'exécutons, nous voyons que `moyenne` vaut 2.



2 ? Si je me rappelle bien de mes cours de maths, c'est plutôt 2.5, non ?

Oui et non. Si nous divisions 5 par 2, nous obtenons bien 2.5. Par contre, ici nous divisons l'entier 5 par l'entier 2 et nous stockons le résultat dans l'entier `moyenne`. Le C# réalise en fait une division entière, c'est-à-dire qu'il prend la partie entière de 2.5, c'est-à-dire 2.

De plus, l'entier `moyenne` est incapable de stocker une valeur contenant des chiffres après la virgule. Il ne prendrait que la partie entière.

Pour avoir 2.5, il faudrait utiliser le code suivant :

```
double moyenne = 5.0 / 2.0;
Console.WriteLine(moyenne);
```

Ici, nous divisons deux `doubles` entre eux et nous stockons le résultat dans un `double`. (Rappelez-vous, le type de données `double` permet de stocker des nombres à virgule.)



Le C# comprend qu'il s'agit de `double` car nous avons ajouté un .0 derrière. Sans ça, il considère que les chiffres sont des entiers.

Les caractères spéciaux dans les chaînes de caractères

En ce qui concerne l'affectation de chaînes de caractères, vous risquez d'avoir des surprises si vous tentez de mettre des caractères spéciaux dans des variables de type `string`. En effet, une chaîne de caractères étant délimitée par des guillemets « `""` », comment faire pour qu'elle puisse contenir des guillemets ?

C'est là qu'intervient le **caractère spécial** \ qui sera à mettre juste devant le guillemet. Par exemple le code suivant :

```
string phrase = "Mon prénom est \"Nicolas\"";
Console.WriteLine(phrase);
```

affichera :

```
Mon prénom est "Nicolas"
```

Si vous avez testé par vous-même l'instruction `Console.WriteLine` et enchaîné plusieurs instructions qui écrivent des lignes, vous avez pu remarquer que nous passions à

la ligne à chaque fois. C'est le rôle de l'instruction `WriteLine` qui affiche la chaîne de caractères et passe à la ligne à la fin de la chaîne de caractères.

Nous pouvons faire la même chose en utilisant le caractère spécial « \n ». Il permet de passer à la ligne à chaque fois qu'il est rencontré. Ainsi, le code suivant :

```
string phrase = "Mon prénom est \"Nicolas\"";
Console.WriteLine(phrase);
Console.WriteLine("Passe\nà\nla\nligne\n\n\n");
```

affichera chaque mot précédé du caractère « \n » sur une ligne différente :

```
Mon prénom est " Nicolas "
Passe
à
la
ligne
```

Vous me diriez qu'on pourrait enchaîner les `Console.WriteLine` et vous auriez raison.

Mais les caractères spéciaux nous permettent de faire d'autres choses comme une tabulation par exemple grâce au caractère spécial « \t ». Le code suivant :

```
Console.WriteLine("Choses à faire :");
Console.WriteLine("\t - Arroser les plantes");
Console.WriteLine("\t - Laver la voiture");
```

permettra d'afficher des tabulations :

```
Choses à faire :
         - Arroser les plantes
         - Laver la voiture
```

Nous avons vu que le caractère \ était un caractère spécial et qu'il permettait de dire au compilateur que nous voulions l'utiliser combiné à la valeur qui le suit, pour avoir une tabulation ou un retour à la ligne. Comment pourrons-nous avoir une chaîne de caractères qui contienne ce fameux caractère ?

Le principe est le même, il suffira de faire suivre ce fameux caractère spécial de lui-même :

```
string fichier = "c:\\repertoire\\fichier.cs";
Console.WriteLine(fichier);
```

La console affichera alors les antislashs :

```
c:\repertoire\fichier.cs
```

Pour ce cas particulier, il est également possible d'utiliser la syntaxe suivante en utilisant le caractère spécial @ devant la chaîne de caractères :

```
string fichier = @"c:\repertoire\fichier.cs"; // contient : c:\repertoire\fichier.cs
```

Bien sûr, nous pouvons stocker des caractères spéciaux dans des variables pour faire par exemple :

```
string sautDeLigne = "\n";
Console.WriteLine("Passer" + sautDeLigne + "à" +
        sautDeLigne + "la" + sautDeLigne + "ligne");
```



Dans ce cas, la variable `sautDeLigne` peut être remplacée par une espèce de variable qui existe déjà dans le framework .NET, à savoir `Environment.NewLine`.

Ce qui permet d'avoir le code suivant :

```
Console.WriteLine("Passer" + Environment.NewLine + "à" +
Environment.NewLine + "la" + Environment.NewLine + "ligne");
```

qui affichera :

```
Passer
à
la
ligne
```



Notez qu'il est possible de passer à la ligne lors de l'écriture d'une instruction C# comme je l'ai fait dans le dernier bout de code afin d'améliorer la lisibilité. N'oubliez pas que c'est le point-virgule qui termine l'instruction.



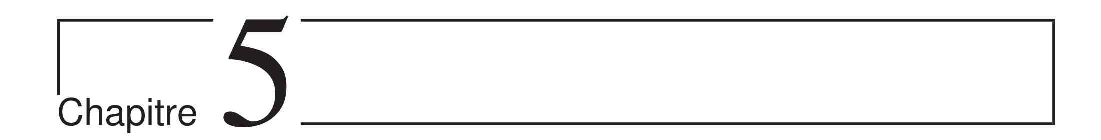
`Environment.NewLine` ? Une espèce de variable ? Qu'est-ce que c'est que cette chose-là ?

En fait, je triche un peu sur les mots. Pour faciliter la compréhension, on peut considérer que `Environment.NewLine` est une variable, au même titre que la variable `sautDeLigne` que nous avons définie. En réalité, c'est un peu plus complexe qu'une variable. Nous découvrirons plus loin de quoi il s'agit vraiment.

En résumé

– Une variable est une zone mémoire permettant de stocker une valeur d'un type particulier.
– Le C# possède plein de types prédéfinis, comme les entiers (`int`), les chaînes de caractères (`string`), etc.

- On utilise l'opérateur = pour affecter une valeur à une variable.
- Il est possible de faire des opérations entre les variables.



Chapitre 5

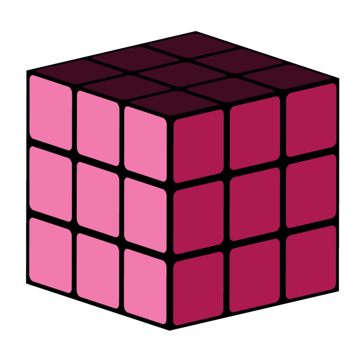
Les instructions conditionnelles

Difficulté : 

Dans nos programmes C#, nous allons régulièrement avoir besoin de faire des opérations en fonction d'un résultat précédent. Par exemple, lors d'un processus de connexion à une application, si le login et le mot de passe sont bons, alors nous pouvons nous connecter, sinon nous afficherons une erreur.

Il s'agit de ce que l'on appelle une condition. Elle est évaluée lors de l'exécution et en fonction de son résultat (vrai ou faux) nous ferons telle ou telle chose.

Bien que relativement court, ce chapitre est très important. N'hésitez pas à le relire et à vous entraîner.



Les opérateurs de comparaison

Une condition se construit grâce à des **opérateurs de comparaison**. On dénombre plusieurs opérateurs de comparaisons, les plus courants sont :

Opérateur	**Description**
==	Égalité
!=	Différence
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal
<=	Inférieur ou égal
&&	ET logique
||	OU logique
!	Négation

Nous allons voir comment les utiliser en combinaison avec les instructions conditionnelles.

L'instruction `if`

L'instruction `if` permet d'exécuter du code si une condition est vraie [1]. Par exemple :

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur");
```

Ici, nous avons une variable contenant le solde de notre compte en banque. Si notre solde est supérieur ou égal à 0 alors nous affichons que le compte est créditeur.

Pour afficher que le compte est débiteur, on pourrait tester si la valeur de la variable est inférieure à 0 et afficher que le compte est débiteur :

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur");
if (compteEnBanque < 0)
    Console.WriteLine("Votre compte est débiteur");
```

Une autre solution est d'utiliser le mot-clé `else`, qui veut dire « sinon » en anglais.

« Si la valeur est vraie, alors on fait quelque chose, sinon, on fait autre chose », ce qui se traduit en C# par :

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
    Console.WriteLine("Votre compte est créditeur");
```

1. Le mot *if* signifie « si » en anglais.

```
else
    Console.WriteLine("Votre compte est débiteur");
```

Il faut bien se rendre compte que l'instruction `if` teste si une valeur est vraie (dans l'exemple précédent la comparaison `compteEnBanque >= 0`).

On a vu rapidement dans les chapitres précédents qu'il existait un type de variable qui permettait de stocker une valeur vraie ou fausse : le type `bool`, autrement appelé **booléen** (*boolean* en anglais). Ainsi, il sera également possible de tester la valeur d'un booléen. L'exemple précédent peut aussi s'écrire :

```
decimal compteEnBanque = 300;
bool estCrediteur = (compteEnBanque >= 0);
if (estCrediteur)
    Console.WriteLine("Votre compte est créditeur");
else
    Console.WriteLine("Votre compte est débiteur");
```

Les parenthèses autour de l'instruction de comparaison sont facultatives, je les ai écrites ici pour clairement identifier que la variable `estCrediteur` va contenir une valeur qui est le résultat de l'opération de comparaison « compte en banque est supérieur ou égal à 0 », en l'occurrence vrai. Voici d'autres exemples pour vous permettre d'appréhender plus précisément le fonctionnement du type `bool` :

```
int age = 30;
bool estAgeDe30Ans = age == 30;
Console.WriteLine(estAgeDe30Ans); // affiche True
bool estSuperieurA10 = age > 10;
Console.WriteLine(estSuperieurA10); // affiche True
bool estDifferentDe30 = age != 30;
Console.WriteLine(estDifferentDe30); // affiche False
```

Un type `bool` peut prendre deux valeurs, **vrai** ou **faux**, qui s'écrivent avec les mots-clés `true` et `false`.

```
bool estVrai = true;
if (estVrai)
    Console.WriteLine("C'est vrai !");
else
    Console.WriteLine("C'est faux !");
```

Il est également possible de combiner les tests grâce aux opérateurs de logique conditionnelle, par exemple && qui correspond à l'opérateur « ET ».

Dans l'exemple qui suit, nous affichons le message de bienvenue uniquement si le login est « Nicolas » ET que le mot de passe est « test ». Si l'un des deux ne correspond pas, nous irons dans l'instruction `else`.

```
string login = "Nicolas";
string motDePasse = "test";
if (login == "Nicolas" && motDePasse == "test")
    Console.WriteLine("Bienvenue Nicolas");
```

```
else
    Console.WriteLine("Login incorrect");
```



Remarquons ici que nous avons utilisé le test d'égalité ==, à ne pas confondre avec l'opérateur d'affection =. C'est une erreur classique de débutant !

D'autres opérateurs de logique existent, nous avons notamment l'opérateur || qui correspond au « OU » logique :

```
if (civilite == "Mme" || civilite == "Mlle")
    Console.WriteLine("Vous êtes une femme");
else
    Console.WriteLine("Vous êtes un homme");
```

L'exemple parle de lui-même ; si la civilité de la personne est Mme ou Mlle, alors nous avons à faire à une femme.

Ici, si la première condition du `if` est vraie alors la deuxième ne sera pas évaluée. C'est un détail ici, mais cela peut s'avérer important dans certaines situations dont une que nous verrons un peu plus loin.

Un autre opérateur très courant est la **négation** que l'on utilise avec l'opérateur « ! ». Par exemple :

```
bool estVrai = true;
if (!estVrai)
    Console.WriteLine("C'est faux !");
else
    Console.WriteLine("C'est vrai !");
```

Ce test pourrait se lire ainsi : si la négation de la variable `estVrai` est vraie, alors on écrira « C'est faux ! » La variable « estVrai » étant égale à `true`, sa négation vaut `false`. Dans cet exemple, le programme nous affichera donc l'instruction correspondant au `else`, à savoir « C'est vrai ! »

Rappelez-vous, nous avons dit qu'une instruction se finissait en général par un point-virgule. Comment cela se fait-il alors qu'il n'y ait pas de point-virgule à la fin du `if` ou du `else` ? Et si nous écrivions l'exemple précédent de cette façon ?

```
bool estVrai = true;
if (!estVrai) Console.WriteLine("C'est faux !");
else Console.WriteLine("C'est vrai !");
```

Ceci est tout à fait valable et permet de voir où s'arrête vraiment l'instruction grâce au point-virgule. Cependant, nous écrivons en général ces instructions de la première façon afin que celles-ci soient plus lisibles. Vous aurez l'occasion de rencontrer dans les chapitres suivants d'autres instructions qui ne se terminent pas obligatoirement par un point-virgule.



Nous verrons dans le chapitre suivant, comment exécuter plusieurs instructions après une instruction conditionnelle en les groupant dans des blocs de code, délimités par des accolades : { et }.

Remarquons enfin qu'il est possible d'enchaîner les tests de manière à traiter plusieurs conditions en utilisant la combinaison `else if`. Cela donne :

```
if (civilite == "Mme")
    Console.WriteLine("Vous êtes une femme");
else if (civilite == "Mlle")
    Console.WriteLine("Vous êtes une femme non mariée");
else if (civilite == "M.")
    Console.WriteLine("Vous êtes un homme");
else
    Console.WriteLine("Je n'ai pas pu déterminer votre civilité
        ");
```

L'instruction `switch`

L'instruction `switch` peut être utilisée lorsqu'une variable prend beaucoup de valeurs. Elle permet de simplifier l'écriture. Ainsi, l'instruction suivante :

```
string civilite = "M.";
if (civilite == "M.")
    Console.WriteLine("Bonjour monsieur");
if (civilite == "Mme")
    Console.WriteLine("Bonjour madame");
if (civilite == "Mlle")
    Console.WriteLine("Bonjour mademoiselle");
```

pourra s'écrire :

```
string civilite = "M.";
switch (civilite)
{
    case "M." :
        Console.WriteLine("Bonjour monsieur");
        break;
    case "Mme":
        Console.WriteLine("Bonjour madame");
        break;
    case "Mlle":
        Console.WriteLine("Bonjour mademoiselle");
        break;
}
```

`switch` commence par **évaluer la variable** qui lui est passée entre parenthèses. Avec le mot-clé `case` on énumère les différents cas possibles pour la variable et on exécute

les instructions correspondantes jusqu'au mot-clé `break` qui signifie que l'on sort du `switch`.

Nous pouvons également indiquer une valeur par défaut en utilisant le mot-clé `default`, ainsi dans l'exemple suivant tout ce qui n'est pas « M. », « Mme » ou « Mlle » donnera l'affichage d'un « Bonjour inconnu » :

```
switch (civilite)
{
    case "M." :
        Console.WriteLine("Bonjour monsieur");
        break;
    case "Mme":
        Console.WriteLine("Bonjour madame");
        break;
    case "Mlle":
        Console.WriteLine("Bonjour mademoiselle");
        break;
    default:
        Console.WriteLine("Bonjour inconnu");
        break;
}
```

Nous pouvons également enchaîner plusieurs cas pour qu'ils fassent la même chose, ce qui reproduit le fonctionnement de l'opérateur logique « OU » (« || »). Par exemple, on pourra remplacer l'exemple suivant :

```
string mois = "Janvier";
if (mois == "Mars" || mois == "Avril" ||  mois == "Mai")
    Console.WriteLine("C'est le printemps");
if (mois == "Juin" || mois == "Juillet" || mois == "Août")
    Console.WriteLine("C'est l'été");
if (mois == "Septembre" || mois == "Octobre" || mois == "
   Novembre")
    Console.WriteLine("C'est l'automne");
if (mois == "Décembre" || mois == "Janvier" || mois == "Février
   ")
    Console.WriteLine("C'est l'hiver");
```

par :

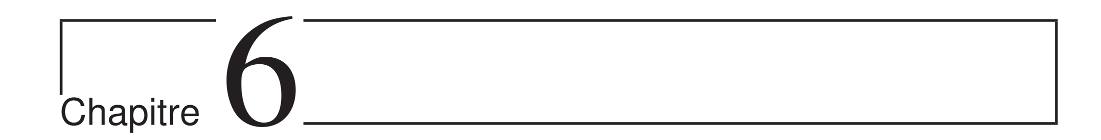
```
switch (mois)
{
    case "Mars":
    case "Avril":
    case "Mai":
        Console.WriteLine("C'est le printemps");
        break;
    case "Juin":
    case "Juillet":
    case "Août":
        Console.WriteLine("C'est l'été");
```

```
        break;
    case "Septembre":
    case "Octobre":
    case "Novembre":
        Console.WriteLine("C'est l'automne");
        break;
    case "Décembre":
    case "Janvier":
    case "Février":
        Console.WriteLine("C'est l'hiver");
        break;
}
```

Ce qui allège quand même l'écriture et la rend beaucoup plus lisible !

En résumé

- Les instructions conditionnelles permettent d'exécuter des instructions seulement si une condition est vérifiée.
- On utilise en général le résultat d'une comparaison dans une instruction conditionnelle.
- Le C# possède beaucoup d'opérateurs de comparaison, comme l'opérateur d'égalité ==, l'opérateur de supériorité >, d'infériorité <, etc.



Chapitre 6

Les blocs de code et la portée d'une variable

Difficulté : 

Nous avons régulièrement utilisé dans le chapitre précédent les accolades ouvrantes et fermantes : { et }. Nous avons rapidement dit que ces accolades servaient à créer des blocs de code.

L'utilisation d'accolades implique également une autre subtilité. Vous l'avez vu dans le titre du chapitre, il s'agit de la portée d'une variable.

Regardons à présent comment cela fonctionne.



Les blocs de code

Les blocs de code permettent de grouper plusieurs instructions qui vont s'exécuter dans le même contexte. Cela peut être le cas par exemple après un `if`, nous pourrions souhaiter effectuer plusieurs instructions. Par exemple :

```
decimal compteEnBanque = 300;
if (compteEnBanque >= 0)
{
    Console.WriteLine("Votre compte est créditeur");
    Console.WriteLine("Voici comment ouvrir un livret ...");
}
else
{
    Console.WriteLine("Votre compte est débiteur");
    Console.WriteLine("N'oubliez pas que les frais de découverts sont de ...");
}
```

Ici, nous enchaînons deux `Console.WriteLine` en fonction du résultat de la comparaison de `compteEnBanque` avec 0.

Les blocs de code seront utiles dès qu'on voudra regrouper plusieurs instructions. C'est le cas pour les instructions conditionnelles mais nous verrons beaucoup d'autres utilisations, comme le `switch` que nous avons vu juste avant, les boucles ou les méthodes que nous allons aborder dans le chapitre suivant.

La portée d'une variable

« C# »... « portée »... on se croirait dans un cours de musique !

En fait, la « portée d'une variable » est la zone de code dans laquelle une variable est utilisable. Elle correspond en général au bloc de code dans lequel est définie la variable.

Ainsi, le code suivant :

```
static void Main(string[] args)
{
    string prenom = "Nicolas";
    string civilite = "M.";
    if (prenom == "Nicolas")
    {
        int age = 30;
        Console.WriteLine("Votre âge est : " + age);
        switch (civilite)
        {
            case "M.":
                Console.WriteLine("Vous êtes un homme de " +
                    age + " ans");
                break;
```

```
                case "Mme":
                    Console.WriteLine("Vous êtes une femme de " +
                        age + " ans");
                    break;
            }
        }
        if (age >= 18)
        {
            Console.WriteLine(prenom + ", vous êtes majeur");
        }
}
```

est incorrect et provoquera une erreur de compilation. En effet, nous essayons d'accéder à la variable `age` en dehors du bloc de code où elle est définie. Nous voyons que cette variable est définie dans le bloc qui est exécuté lorsque le test d'égalité du prénom avec la chaîne « Nicolas » est vrai alors que nous essayons de la comparer à « 18 » dans un endroit où elle n'existe plus.

Nous pouvons utiliser `age` dans tout le premier `if`, et dans les sous-blocs de code, comme c'est le cas dans le sous-bloc du `switch`, mais pas en dehors du bloc de code dans lequel la variable est définie. Ainsi, la variable `prenom` est accessible dans le dernier `if` car elle a été définie dans un bloc père.

Vous noterez qu'ici, la complétion nous est utile. En effet, Visual C# Express propose de compléter le nom de la variable dans un bloc où elle est accessible. Dans un bloc où elle ne l'est pas, la complétion ne nous la propose pas. Comme Visual C# Express est malin, si la complétion ne propose pas ce que vous souhaitez, c'est que vous n'y avez pas le droit ! Cela peut être parce que la portée ne vous l'autorise pas. Pour corriger l'exemple précédent, il faut déclarer la variable `age` au même niveau que la variable `prenom`.



Ok, mais alors, pourquoi on ne déclarerait pas tout au début une bonne fois pour toutes ? Cela éviterait ces erreurs. . . non ?

Eh non, ce n'est pas possible ! Généralement, l'utilisation de variables accessibles de partout est une mauvaise pratique de développement. Même si on peut avoir un équivalent en C#, il faut se rappeler que plus une variable est utilisée dans la plus petite portée possible, mieux elle sera utilisée et plus elle sera pertinente. Essayez donc de déterminer le bloc de code minimal où l'utilisation de la variable est adaptée.

En résumé

- Un bloc de code permet de regrouper des instructions qui commencent par { et qui finissent par }.
- Une variable définie à l'intérieur d'un bloc de code aura pour portée ce bloc de code.

Chapitre 7

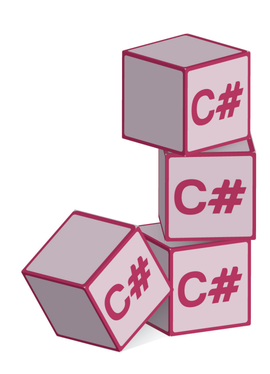
Les méthodes

Difficulté : 

Élément indispensable de tout programme informatique, une méthode regroupe un ensemble d'instructions, pouvant prendre des paramètres et pouvant renvoyer une valeur. Lors de vos développements, vous allez avoir besoin de créer beaucoup de méthodes.

Nous allons découvrir les méthodes dans ce chapitre mais nous y reviendrons petit à petit tout au long de l'ouvrage et vous aurez ainsi l'occasion d'approfondir vos connaissances.

Vous pourrez trouver de temps en temps le mot « **fonction** » à la place du mot « **méthode** ». Cela signifie la même chose. C'est une relique du passé correspondant à un ancien mode de développement qui s'utilise de moins en moins, de même que le terme « procédure » qui est encore plus vieux !



Créer une méthode

Le but de la méthode est de factoriser du code afin d'éviter d'avoir à répéter sans cesse le même code et ceci pour deux raisons essentielles :

- déjà parce que l'homme est un être paresseux qui utilise son intelligence pour éviter tout travail inutile ;
- ensuite parce que si jamais il y a quelque chose à corriger dans ce bout de code et s'il est dupliqué à plusieurs endroits, nous allons devoir faire une correction dans tous ces endroits. Si le code est factorisé à un unique endroit, nous ferons une unique correction. Oui, oui, encore la paresse mais cela permet aussi d'éviter d'oublier un bout de code dans un endroit caché.

Ce souci de factorisation est connu comme le principe **DRY**, qui est l'acronyme des mots anglais *Don't Repeat Yourself* (Ne vous répétez pas). Le but est de ne jamais (à quelques exceptions près évidemment...) avoir à réécrire la même ligne de code.

Par exemple, imaginons quelques instructions qui s'occupent d'écrire un message de bienvenue avec le nom de l'utilisateur. Le code C# pourrait être :

```
Console.WriteLine("Bonjour Nicolas");
Console.WriteLine("-------" + Environment.NewLine);
Console.WriteLine("\tBienvenue dans le monde merveilleux du C#"
    );
```



Dans l'instruction `Console.WriteLine` que nous utilisons régulièrement, `WriteLine` est une méthode.

Si plus tard on veut réafficher le message de bienvenue, il faudra réécrire ces quatre lignes de code à moins que nous utilisions une méthode :

```
static void AffichageBienvenue()
{
    Console.WriteLine("Bonjour Nicolas");
    Console.WriteLine("-------" + Environment.NewLine);
    Console.WriteLine("\tBienvenue dans le monde merveilleux du
        C#");
}
```

Dans l'exemple précédent, je définis une méthode qui s'appelle `AffichageBienvenue`.

L'instruction :

```
static void AffichageBienvenue()
```

est ce qu'on appelle la signature de la méthode. Elle nous renseigne sur les paramètres de la méthode et sur ce qu'elle va renvoyer. Le mot-clé `void` signifie que la méthode ne renvoie rien. Les parenthèses vides à la fin de la signature indiquent que la méthode n'a pas de paramètre.



C'est la forme de la méthode la plus simple possible.

Le mot-clé `static` ne nous intéresse pas pour l'instant, mais sachez qu'il sert à indiquer que la méthode est toujours disponible et prête à être utilisée. Dans ce contexte, il est obligatoire. Nous y reviendrons.

En dessous de la signature de la méthode, nous retrouvons les accolades. Elles permettent de délimiter la méthode. Le bloc de code ainsi formé constitue ce qu'on appelle le « corps de la méthode ».

En résumé, pour déclarer une méthode, nous aurons :

```
Signature de la méthode
{
    Corps de la méthode
}
```

Nous pouvons désormais appeler cette méthode (c'est-à-dire l'exécuter), dans notre programme grâce à son nom. Par exemple, ici je l'appelle très facilement deux fois de suite :

```
static void Main(string[] args)
{
    AffichageBienvenue();
    AffichageBienvenue();
}

static void AffichageBienvenue()
{
    Console.WriteLine("Bonjour Nicolas");
    Console.WriteLine("-------" + Environment.NewLine);
    Console.WriteLine("\tBienvenue dans le monde merveilleux du
        C#");
}
```

Et tout ça, sans effort ! C'est quand même plus simple et plus clair, non ?

La méthode spéciale `Main()`

La signature du premier bloc ci-dessus ne vous rappelle rien ? Mais si, elle ressemble beaucoup à celle de la méthode que nous avons créée précédemment.



Il s'agit d'une méthode spéciale, la méthode `Main()`.

Elle a été générée par Visual C# Express lorsque nous avons créé le projet **Console**.

Cette méthode est en fait le point d'entrée de l'application, c'est-à-dire que quand le CLR tente d'exécuter notre application, il recherche cette méthode afin de pouvoir commencer à exécuter des instructions à partir d'elle. S'il ne la trouve pas, alors, il ne pourra pas exécuter notre application. C'est pour cela qu'il est important que cette méthode soit accessible de partout ; rappelez-vous, c'est le rôle du mot-clé `static` que nous aurons l'occasion d'étudier plus en détail ultérieurement. Visual C# Express nous garde bien de cette erreur. En effet, si vous supprimez cette méthode (ou que vous enlevez le mot-clé `static`) et que vous tentez de compiler notre application, vous aurez le message d'erreur suivant :

```
Erreur  1       Le programme 'C:\Users\Nico\Documents\Visual
    Studio 2010\Projects\C\#\MaPremiereApplication\
    MaPremiereApplication\obj\x86\Release\MaPremiereApplication.
    exe' ne contient pas une méthode 'Main' statique appropriée
    pour un point d'entrée
```

Le message d'erreur est clair. Il a besoin d'une méthode `Main()` pour démarrer.



Voilà pour cette méthode spéciale `Main()`. Elle est indispensable dans tout programme exécutable et c'est par-là que le programme démarre.

Les lecteurs attentifs auront remarqué que cette méthode possède certains éléments dans la signature, entre les parenthèses... Des paramètres ! Découvrons-les dans la prochaine section...

Paramètres d'une méthode

Super, nous savons créer des méthodes. Nous allons pouvoir créer une méthode qui permet de souhaiter la bienvenue à la personne qui vient de se connecter à notre application. Si c'est Nicolas qui vient de se connecter, nous allons appeler la méthode `AffichageBienvenueNicolas()`. Si c'est Jérémie, nous appellerons la méthode `AffichageBienvenueJeremie()`, etc.

```
static void AffichageBienvenueNicolas()
{
    Console.WriteLine("Bonjour Nicolas");
    Console.WriteLine("-------" + Environment.NewLine);
    Console.WriteLine("\tBienvenue dans le monde merveilleux du
        C#");
}

static void AffichageBienvenueJeremie()
{
    Console.WriteLine("Bonjour Jérémie");
    Console.WriteLine("-------" + Environment.NewLine);
```

```
    Console.WriteLine("\tBienvenue dans le monde merveilleux du
        C#");
}
```

Finalement, ce n'est pas si pratique que ça... Alors que nous venions juste d'évoquer le principe **DRY**, nous nous retrouvons avec deux méthodes quasiment identiques qui ne diffèrent que d'un tout petit détail.

C'est là qu'interviennent les paramètres de méthode. Nous l'avons évoqué au paragraphe précédent, il est possible de passer des paramètres à une méthode. Ainsi, nous pourrons utiliser les valeurs de ces paramètres dans le corps de nos méthodes, les méthodes en deviendront d'autant plus génériques.

Dans notre exemple d'affichage de message de bienvenue, il est évident que le nom de l'utilisateur sera un paramètre de la méthode.

Les paramètres s'écrivent à l'intérieur des parenthèses qui suivent le nom de la méthode. Nous devons indiquer le type du paramètre ainsi que le nom de la variable qui le représentera au sein de la méthode.

Il est possible de passer plusieurs paramètres à une méthode, on les séparera avec une virgule. Par exemple :

```
static void DireBonjour(string prenom, int age)
{
    Console.WriteLine("Bonjour " + prenom);
    Console.WriteLine("Vous avez " + age + " ans");
}
```

Ici, la méthode `DireBonjour` prend en paramètres une chaîne de caractères `prenom` et un entier `age`. La méthode affiche « Bonjour » ainsi que le contenu de la variable `prenom`. De même, juste en dessous, elle affiche l'âge qui a été passé en paramètre.

Nous pourrons appeler cette méthode de cette façon, depuis la méthode `Main()` :

```
static void Main(string[] args)
{
    DireBonjour("Nicolas", 30);
    DireBonjour("Jérémie", 20);
}
```

Ce qui permettra d'afficher facilement un message de bienvenue personnalisé :

```
Bonjour Nicolas
Vous avez 30 ans
Bonjour Jérémie
Vous avez 20 ans
```

Bien sûr, il est obligatoire de fournir en paramètre d'une méthode une variable de même type que le paramètre attendu. Dans le cas contraire, le compilateur est incapable de mettre la donnée qui a été passée dans le paramètre. D'ailleurs, si vous ne fournissez pas le bon paramètre, vous avez droit à une erreur de compilation.

Par exemple, si vous appelez la méthode avec les paramètres suivants :

```
DireBonjour(10, 10);
```

Vous aurez l'erreur de compilation suivante :

```
impossible de convertir de 'int' en 'string'
```

Il est évidemment possible de passer des variables à une méthode, cela fonctionne de la même façon :

```
string prenom = "Nicolas";
DireBonjour(prenom, 30);
```

Nous allons revenir plus en détail sur ce qu'il se passe exactement lorsque nous aborderons le chapitre sur le mode de passage des paramètres.

Vous voyez, cela ressemble beaucoup à ce que nous avons déjà fait avec la méthode `Console.WriteLine()`. Facile, non ?

La méthode `Console.WriteLine` fait partie de la bibliothèque du framework .NET. Elle est utilisée pour écrire des chaînes de caractères, des nombres ou plein d'autres choses sur la console. Le framework .NET contient énormément de méthodes utilitaires de toutes sortes. Nous y reviendrons un peu plus tard.

Vous aurez peut-être remarqué un détail : nous avons préfixé toutes nos méthodes du mot-clé `static`. J'ai dit que c'était obligatoire dans notre contexte, pour être plus précis, c'est parce que la méthode `Main()` est statique que nous sommes obligés de créer des méthodes statiques. On a dit que la méthode `Main()` était obligatoirement statique parce qu'elle devait être accessible de partout afin que le CLR puisse trouver le point d'entrée de notre programme. Or, une méthode statique ne peut appeler que des méthodes statiques, c'est pour cela que nous sommes obligés (pour l'instant) de préfixer nos méthodes par le mot-clé `static`. Nous décrirons ce que recouvre exactement le mot-clé `static` dans la partie suivante.

Retour d'une méthode

Une méthode peut aussi renvoyer une valeur, par exemple un calcul. C'est souvent d'ailleurs son utilité première.

On pourrait imaginer par exemple une méthode qui calcule la longueur de l'hypoténuse à partir des deux côtés d'un triangle rectangle. Sachant que $a^2 + b^2 = c^2$, nous pouvons imaginer une méthode qui prenne en paramètres la longueur des deux côtés, fasse la somme de leurs carrés et renvoie la racine carrée du résultat. C'est ce que fait la méthode suivante :

```
static double LongueurHypotenuse(double a, double b)
{
    double sommeDesCarres = a * a + b * b;
    double resultat = Math.Sqrt(sommeDesCarres);
```

```
    return resultat;
}
```

Vous aurez remarqué que la signature de la méthode commence par le mot-clé `double`, qui indique qu'elle va nous renvoyer une valeur de type `double`. Comme on l'a vu, `double a` et `double b` sont deux paramètres de la méthode et sont du type `double`.

La méthode `Math.Sqrt` est une méthode du framework .NET, au même titre que la méthode `Console.WriteLine`, qui permet de renvoyer la racine carrée d'un nombre. Elle prend en paramètre un `double` et nous retourne une valeur de type `double` également qui correspond à la racine carrée du paramètre. C'est tout naturellement que nous stockons ce résultat dans une variable grâce à l'opérateur d'affectation =.

À la fin de la méthode, le mot-clé `return` indique que la méthode renvoie la valeur à la méthode qui l'a appelée. Ici, nous renvoyons le résultat.

Cette méthode pourra s'utiliser ainsi :

```
static void Main(string[] args)
{
    double valeur = LongueurHypotenuse(1, 3);
    Console.WriteLine(valeur);
    valeur = LongueurHypotenuse(10, 10);
    Console.WriteLine(valeur);
}
```

Comme précédemment, nous utilisons une variable pour stocker le résultat de l'exécution de la méthode :

```
3,16227766016838
14,142135623731
```

Notez qu'il est également possible de se passer d'une variable intermédiaire pour stocker le résultat. Ainsi, nous pourrons par exemple écrire :

```
Console.WriteLine("Le résultat est : " + LongueurHypotenuse(1,
    3));
```

Avec cette écriture le résultat renvoyé par la méthode `LongueurHypotenuse` est directement concaténé à la chaîne « Le résultat est : » et passé en paramètre à la méthode `Console.WriteLine`.

Remarquez qu'on a fait l'opération : « a*a » pour mettre « a » au carré. On aurait également pu faire `Math.Pow(a, 2)` qui permet de faire la même chose à la différence que `Pow` permet de mettre à la puissance que l'on souhaite. Ainsi, `Math.Pow(a, 3)` permet de mettre « a » au cube.

Il faut savoir que le mot-clé `return` peut apparaître à n'importe quel endroit de la méthode. Il interrompt alors l'exécution de celle-ci et renvoie la valeur passée. Ce mot-clé est obligatoire, sans cela la méthode ne compile pas. Il est également primordial que tous les chemins possibles d'une méthode renvoient quelque chose. Les chemins sont déterminés par les instructions conditionnelles que nous avons vues précédemment.

Ainsi, l'exemple suivant est correct :

```
static string Conjugaison(string genre)
{
    if (genre == "homme")
        return "é";
    else
        return "ée";
}
```

car peu importe la valeur de la variable `genre`, la méthode renverra une chaîne.

Alors que celui-ci :

```
static string Conjugaison(string genre)
{
    if (genre == "homme")
        return "é";
    else
    {
        if (genre == "femme")
            return "ée";
    }
}
```

est incorrect. En effet, que renvoie la méthode si la variable `genre` contient autre chose que « homme » ou « femme » ?

En général, Visual C# Express nous indiquera qu'il détecte un problème avec une erreur de compilation.

Nous pourrons corriger ceci avec, par exemple :

```
static string Conjugaison(string genre)
{
    if (genre == "homme")
        return "é";
    else
    {
        if (genre == "femme")
            return "ée";
    }
    return "";
}
```



À noter que `""` correspond à une chaîne vide et peut également s'écrire `string.Empty`.

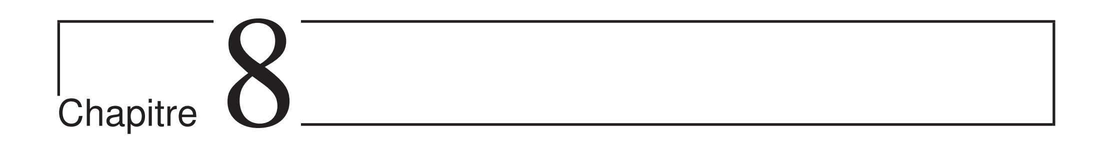
Nous avons vu dans le début du chapitre qu'il était possible de créer des méthodes qui ne retournent rien. Dans ce cas, on peut utiliser le mot-clé `return` sans valeur qui le suit pour stopper l'exécution de la méthode. Par exemple :

```
static void Bonjour(string prenom)
{
    if (prenom == "inconnu")
        return;
    Console.WriteLine("Bonjour " + prenom);
}
```

Ainsi, si la variable `prenom` vaut « inconnu », alors nous quittons la méthode `Bonjour` et l'instruction `Console.WriteLine` ne sera pas exécutée.

En résumé

- Une méthode regroupe un ensemble d'instructions pouvant prendre des paramètres et pouvant renvoyer une valeur.
- Les paramètres d'une méthode doivent être utilisés avec le bon type.
- Une méthode qui ne renvoie rien est préfixée du mot-clé `void`.
- Le point d'entrée d'un programme est la méthode statique `Main()`.
- Le mot-clé `return` permet de renvoyer une valeur du type de retour de la méthode, à l'appelant de cette méthode.



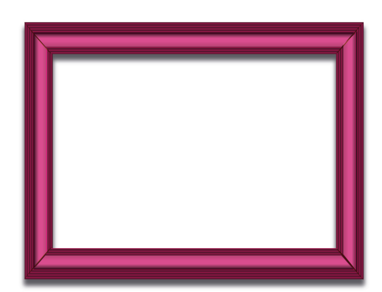
Chapitre 8

Tableaux, listes et énumérations

Difficulté : 

Dans les chapitres précédents, nous avons pu utiliser les types de base du framework .NET, comme `int`, `string`, `double`, etc. Nous allons découvrir ici d'autres types qui vont s'avérer très utiles dans la construction de nos applications informatiques.

Une fois bien maîtrisés, vous ne pourrez plus vous en passer !



Les tableaux

Voici le premier nouveau type que nous allons étudier, le type tableau. En déclarant une variable de type tableau, nous allons en fait utiliser une variable qui contient une suite de variables du même type. Prenons cet exemple :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

Nous définissons ici un tableau de chaînes de caractères qui contient 7 chaînes de caractères : les jours de la semaine. Ne faites pas attention à l'opérateur `new` pour l'instant, nous y reviendrons plus tard ; il permet simplement de créer le tableau. Les crochets « [] » qui suivent le nom du type permettent de signaler au compilateur que nous souhaitons utiliser un tableau de ce type-là, ici le type `string`.

Un tableau, c'est un peu comme une armoire dans laquelle on range des variables. Chaque variable est posée sur une étagère. Pour accéder à la variable qui est posée sur une étagère, on utilise le nom de l'armoire et on indique l'indice de l'étagère où est stockée la variable, en utilisant des crochets [] :

```
Console.WriteLine(jours[3]); // affiche Jeudi
Console.WriteLine(jours[0]); // affiche Lundi
Console.WriteLine(jours[10]); // provoque une erreur d'exécution car l'indice n'existe pas
```



Le premier élément du tableau se situe à l'indice 0 et le dernier se situe à l'indice « taille du tableau – 1 », c'est-à-dire 6 dans notre exemple. Si on tente d'accéder à un indice qui n'existe pas, l'application lèvera une erreur.

Il est possible de parcourir l'ensemble d'un tableau avec l'instruction suivante :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
for (int i = 0; i < jours.Length; i++)
{
    Console.WriteLine(jours[i]);
}
```

Nous parcourons ici les éléments de 0 à taille-1 et nous affichons l'élément du tableau correspondant à l'indice en cours. Ce qui nous donne :

```
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
```

Revenons à présent sur la déclaration du tableau :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

Cette écriture permet de créer un tableau qui contient 7 éléments et d'affecter une valeur à chaque élément du tableau. Il s'agit en fait ici d'une écriture simplifiée. Cette écriture est équivalente à celle-ci :

```
string[] jours = new string[7];
jours[0] = "Lundi";
jours[1] = "Mardi";
jours[2] = "Mercredi";
jours[3] = "Jeudi";
jours[4] = "Vendredi";
jours[5] = "Samedi";
jours[6] = "Dimanche";
```

qui est beaucoup plus verbeuse, mais d'un autre côté, plus explicite.

La première instruction crée un tableau qui peut contenir 7 éléments. C'est la taille du tableau, elle ne peut pas changer. Chaque instruction suivante affecte une valeur à un indice du tableau. Rappelez-vous, un tableau commence à l'indice 0 et va jusqu'à l'indice taille – 1.

Il est possible facilement de faire des opérations sur un tableau, comme un tri. On pourra utiliser la méthode `Array.Sort()`. Par exemple :

```
Array.Sort(jours);
```

Avec cette instruction, le tableau sera classé par ordre alphabétique. Vous aurez l'occasion de voir d'autres méthodes dans des chapitres ultérieurs. Ainsi, le code suivant :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
Array.Sort(jours);
for (int i = 0; i < jours.Length; i++)
{
    Console.WriteLine(jours[i]);
}
```

produira un classement alphabétique des entrées du tableau :

```
Dimanche
Jeudi
Lundi
Mardi
Mercredi
Samedi
Vendredi
```

Ce qui est très inutile !

Le tableau `jours` est ce que l'on appelle un tableau à une dimension. Il est également possible de créer des tableaux à N dimensions, il est cependant assez rare de dépasser

deux dimensions. C'est surtout utile lorsqu'on manipule des matrices.

Nous n'étudierons pas les tableaux à plus d'une dimension dans ce livre car ils risquent de vraiment peu vous servir dans vos premières applications. Par contre, le type « liste », que nous allons voir tout de suite, vous servira abondamment !

Les listes

Un autre type que nous allons beaucoup utiliser est la liste. Nous allons voir comment ce type fonctionne mais sans en faire une étude exhaustive car elle pourrait être bien longue et ennuyeuse. Regardons cet exemple :

```
List<int> chiffres = new List<int>(); // création de la liste
chiffres.Add(8); // chiffres contient 8
chiffres.Add(9); // chiffres contient 8, 9
chiffres.Add(4); // chiffres contient 8, 9, 4

chiffres.RemoveAt(1); // chiffres contient 8, 4

foreach (int chiffre in chiffres)
{
    Console.WriteLine(chiffre);
}
```

La première ligne permet de créer la liste. Nous reviendrons sur cette instruction un peu plus loin dans ce chapitre. Il s'agit d'une liste d'entiers.

Nous ajoutons des entiers à la liste grâce à la méthode `Add()`. Nous ajoutons en l'occurrence les entiers 8, 9 et 4.

La méthode `RemoveAt()` permet de supprimer un élément en utilisant son indice, ici nous supprimons le deuxième entier, c'est-à-dire 9.



Comme les tableaux, le premier élément de la liste commence à l'indice 0.

Après cette instruction, la liste contient les entiers 8 et 4.

Enfin, nous parcourons les éléments de la liste grâce à l'instruction `foreach`. Nous y reviendrons en détail lors du chapitre sur les boucles. Pour l'instant, nous avons juste besoin de comprendre que nous affichons tous les éléments de la liste, comme ci-dessous :

```
8
4
```

Les lecteurs assidus auront remarqué que la construction de la liste est un peu particulière. On observe en effet l'utilisation de chevrons <> pour indiquer le type de la liste.

Pour avoir une liste d'entier, il suffit d'indiquer le type `int` à l'intérieur des chevrons. Ainsi, il ne sera pas possible d'ajouter autre chose qu'un entier dans cette liste. Par exemple, l'instruction suivante provoque une erreur de compilation :

```
List<int> chiffres = new List<int>(); // création de la liste
chiffres.Add("chaîne"); // ne compile pas
```

Évidemment, on ne peut pas ajouter des choux à une liste de carottes !

De la même façon, si l'on souhaite créer une liste de chaîne de caractères, nous pourrons utiliser le type `string` à l'intérieur des chevrons :

```
List<string> chaînes = new List<string>(); // création de la
    liste
chaînes.Add("chaîne"); // compilation OK
chaînes.Add(1); // compilation KO, on ne peut pas ajouter un
    entier dans une liste de chaînes de caractères
```

Les listes possèdent des méthodes bien pratiques qui permettent toutes sortes d'opérations. Par exemple, la méthode `IndexOf()` permet de rechercher un élément dans la liste et de renvoyer son indice.

```
List<string> jours = new List<string>();
jours.Add("Lundi");
jours.Add("Mardi");
jours.Add("Mercredi");
jours.Add("Jeudi");
jours.Add("Vendredi");
jours.Add("Samedi");
jours.Add("Dimanche");

int indice = jours.IndexOf("Mercredi"); // indice vaut 2
```

Nous aurons l'occasion de voir d'autres utilisations de méthodes de la liste dans les chapitres suivants.

La liste que nous venons de voir (`List<>`) est en fait ce que l'on appelle **un type générique**. Nous n'allons pas rentrer dans le détail de ce qu'est un type générique pour l'instant, mais il faut juste savoir qu'un type générique permet d'être spécialisé par un type concret. Pour notre liste, cette généricité permet d'indiquer de quel type est la liste, une liste d'entiers ou une liste de chaînes de caractères, etc.

Ne vous inquiétez pas si tout ceci n'est pas parfaitement clair, nous reviendrons plus en détail sur les génériques dans un chapitre ultérieur. Le but ici est de commencer à se familiariser avec le type `List<>` que nous utiliserons régulièrement et les exemples que nous verrons permettront d'appréhender les subtilités de ce type.

À noter qu'il existe également une écriture simplifiée des listes. En effet, il est possible de remplacer :

```
List<string> jours = new List<string>();
jours.Add("Lundi");
jours.Add("Mardi");
```

```
jours.Add("Mercredi");
jours.Add("Jeudi");
jours.Add("Vendredi");
jours.Add("Samedi");
jours.Add("Dimanche");
```

par

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

un peu comme pour les tableaux.

Liste ou tableau ?

Vous aurez remarqué que les deux types, tableau et liste, se ressemblent beaucoup. Essayons de voir ce qui les différencie afin de savoir quel type choisir entre les deux.

En fait, une des grosses différences est que le tableau peut être multidimensionnel. C'est un point important mais dans le cadre de vos premières applications, il est relativement rare d'avoir à s'en servir. Cela sert le plus souvent dans le développement de jeux ou lorsque l'on souhaite faire des calculs 3D.

Par contre, le plus important pour nous est que le type tableau est de taille fixe alors que la liste est de taille variable. On peut ajouter sans problème un nouvel élément à une liste grâce à la méthode `Add()`. De même, on peut supprimer des éléments avec la méthode `Remove` alors qu'avec un tableau, on peut seulement remplacer les valeurs existantes et il n'est pas possible d'augmenter sa capacité.

Globalement, vous verrez que dans vos applications, vous utiliserez plutôt les listes, par exemple pour afficher une liste de produits, une liste de clients, etc.

Gardez quand même dans un coin de l'esprit les tableaux, ils pourront vous aider dans des situations précises.

Les énumérations

Un type particulier que nous allons également utiliser est l'énumération. Cela correspond comme son nom l'indique à une énumération de valeurs.

Par exemple, il pourrait être très facile de représenter les jours de la semaine dans une énumération plutôt que dans un tableau.

On définit l'énumération de cette façon, grâce au mot-clé `enum` :

```
enum Jours
{
    Lundi,
    Mardi,
    Mercredi,
```

```
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

À noter qu'on ne peut pas définir cette énumération n'importe où, pour l'instant, contentons-nous de la définir en dehors de la méthode `Main()`.

Pour être tout à fait précis, une énumération est un type dont toutes les valeurs définies sont des entiers. La première vaut 0, et chaque valeur suivante prend la valeur précédente augmentée de 1. C'est-à-dire que Lundi vaut 0, Mardi vaut 1, etc.

Il est possible de forcer des valeurs à toutes ou certaines valeurs de l'énumération, les valeurs non forcées prendront la valeur précédente augmentée de 1 :

```
enum Jours
{
    Lundi = 5, // lundi vaut 5
    Mardi, // mardi vaut 6
    Mercredi = 9, // mercredi vaut 9
    Jeudi = 10, // jeudi vaut 10
    Vendredi, // vendredi vaut 11
    Samedi, // samedi vaut 12
    Dimanche = 20 // dimanche vaut 20
}
```

Mais, à part pour enregistrer une valeur dans une base de données, il est rare de manipuler les énumérations comme des entiers car le but de l'énumération est justement d'avoir une liste exhaustive et fixée de valeurs constantes. Le code s'en trouve plus clair, plus simple et plus lisible.

Le fait de définir une telle énumération revient en fait à enrichir les types que nous avons à notre disposition, comme les entiers ou les chaînes de caractères (`int` ou `string`). Ce nouveau type s'appelle `Jours`.

Nous pourrons définir une variable du type `Jours` de la même façon qu'avec un autre type :

```
Jours jourDeLaSemaine;
```

La seule différence c'est que les valeurs qu'il est possible d'affecter à notre variable sont figées et font partie des valeurs définies dans l'énumération.

Pour pouvoir accéder à un élément de l'énumération, il faudra utiliser le nom de l'énumération suivi de l'opérateur point « . », suivi encore de la valeur de l'énumération choisie. Par exemple :

```
Jours lundi = Jours.Lundi;
Console.WriteLine(lundi);
```

soit dans notre programme :

```
class Program
```

```
{
    enum Jours
    {
        Lundi,
        Mardi,
        Mercredi,
        Jeudi,
        Vendredi,
        Samedi,
        Dimanche
    }

    static void Main(string[] args)
    {
        Jours lundi = Jours.Lundi;
        Console.WriteLine(lundi);
    }
}
```

La console nous affichera :

```
Lundi
```

Nous pourrons également nous servir des énumérations pour faire des tests de comparaison, comme par exemple :

```
if (jourDeLaSemaine == Jours.Dimanche || jourDeLaSemaine ==
    Jours.Samedi)
{
    Console.WriteLine("Bon week-end");
}
```

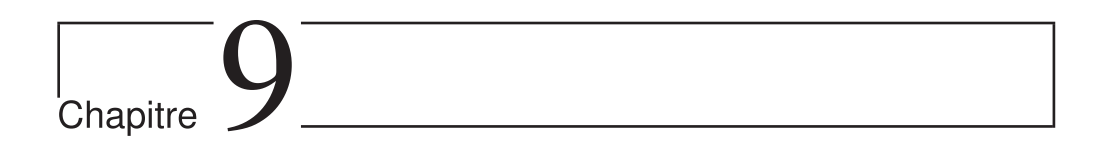
Sachez que le framework .NET utilise beaucoup les énumérations. Il est important de savoir les manipuler.



Vous aurez peut-être remarqué que lorsqu'on affiche la valeur d'une énumération, la console nous affiche le nom de l'énumération et non pas sa valeur entière. Ça peut paraître étrange, mais c'est parce que le C# fait un traitement particulier dans le cadre d'une énumération à l'affichage.

En résumé

- Un tableau est un type évolué pouvant contenir une séquence d'autres types, comme un tableau d'entiers ou un tableau de chaînes de caractères.
- Une liste est un type complexe un peu plus souple que le tableau permettant d'avoir une liste de n'importe quel type.
- Une énumération s'utilise lorsque l'on veut créer un type possédant plusieurs valeurs fixes, comme les jours de la semaine.



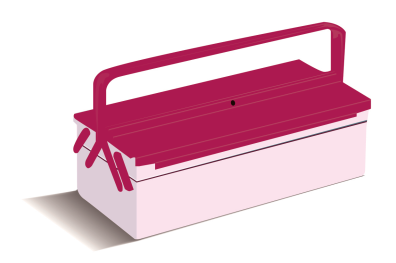
Chapitre 9

Utiliser le framework .NET

Difficulté : 

Comme on l'a déjà évoqué, le framework .NET est une énorme boîte à outils qui contient beaucoup de méthodes permettant de construire toutes sortes d'applications. Nous allons avoir besoin régulièrement d'utiliser les éléments du framework .NET pour réaliser nos applications. Il est donc grand temps d'apprendre à savoir le manipuler !

Rentrons tout de suite dans le vif du sujet.



L'instruction using

Nous allons sans arrêt solliciter la puissance du framework .NET. Par exemple, nous pouvons lui demander de nous donner la date courante.

Pour ce faire, on utilisera l'instruction :

```
Console.WriteLine(DateTime.Now);
```

Ce qui se passe ici, c'est que nous demandons à notre application l'affichage de la propriété `Now` de l'objet `DateTime`. Nous allons revenir en détail sur ce que sont des propriétés et des objets, considérez pour l'instant qu'ils correspondent simplement à une instruction qui nous fournit la date du moment :

```
09/08/2011 23:47:58
```

En fait, pour accéder à la date courante, on devrait normalement écrire :

```
System.Console.WriteLine(System.DateTime.Now);
```

En effet, les objets `DateTime` et `Console` se situent dans l'espace de nom `System`.

Un espace de nom [1] correspond à un endroit où l'on range des méthodes et des objets. Il est caractérisé par des mots séparés par des points « . ».

C'est un peu comme des répertoires, nous pouvons dire que le fichier `DateTime` est rangé dans le répertoire « System » et, quand nous souhaitons y accéder, nous devons fournir l'emplacement complet du fichier, à savoir `System.DateTime`.

Cependant, plutôt que d'écrire le chemin complet à chaque fois, il est possible de dire : « OK, maintenant, à chaque fois que je vais avoir besoin d'accéder à une fonctionnalité, va la chercher dans l'espace de nom `System`. Si elle s'y trouve, utilise-la ».

C'est ce qui se passe grâce à l'instruction suivante :

```
using System;
```

qui a été générée par Visual C# Express au début du fichier.

Elle indique au compilateur que nous allons utiliser l'espace de nom `System` dans notre page. Ainsi, il n'est plus obligatoire de préfixer l'accès à `DateTime` par `System`. C'est une espèce de raccourci. Pour conserver l'analogie avec les répertoires et les fichiers, on peut dire que nous avons ajouté le répertoire « System » dans le path.

Pour résumer, l'instruction :

```
System.Console.WriteLine(System.DateTime.Now);
```

est équivalente aux deux instructions :

```
using System;
```

et

1. En anglais, *namespace*

```
Console.WriteLine(DateTime.Now);
```

Sachant qu'elles ne s'écrivent pas côte à côte. En général, on met l'instruction `using` en en-tête du fichier `.cs`, comme ce qu'a fait Visual C# Express lorsqu'il a généré le fichier. L'autre instruction est à positionner à l'endroit où nous souhaitons qu'elle soit exécutée.



Si le `using System` est absent, la complétion automatique de Visual C# Express ne vous proposera pas le mot `DateTime`. C'est un bon moyen de se rendre compte qu'il manque la déclaration de l'utilisation de l'espace de nom.

À noter que dans ce cas-là, si Visual C# Express reconnaît l'instruction mais que l'espace de nom n'est pas inclus, il le propose en soulignant le début du mot `DateTime` d'un petit trait bleu et blanc, comme l'illustre la figure 9.1.

```
class Program
{
    static void Main(string[] args)
    {
        DateTime
    }
}
```

FIGURE 9.1 – Le trait bleu permet d'indiquer que l'IDE sait résoudre l'import de l'espace de nom

Un clic droit sur le mot permettra d'ouvrir un menu déroulant, de choisir `Résoudre` et d'importer le `using` correspondant automatiquement (voir figure 9.2).

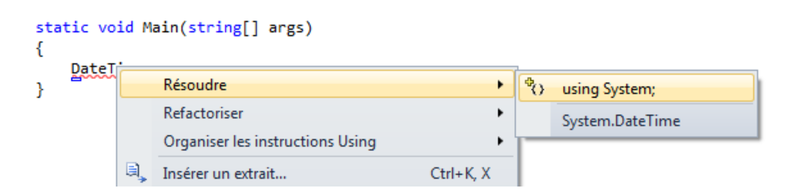


FIGURE 9.2 – Importer l'espace de nom automatiquement

La bibliothèque de classes .NET

Vous aurez l'occasion de découvrir que le framework .NET fourmille d'espaces de noms contenant plein de méthodes de toutes sortes permettant de faire un peu tout et n'importe quoi. Une vraie caverne d'Ali Baba !

Parmi les nombreuses fonctionnalités du framework .NET, nous avons une méthode qui nous permet de récupérer l'utilisateur courant (au sens « utilisateur Windows »), on pourra par exemple utiliser l'instruction suivante :

```
Console.WriteLine(System.Environment.UserName);
```

qui affichera l'utilisateur Windows dans la console :

```
Nico
```

Ou, comme vous le savez désormais, on pourra utiliser conjointement :

```
using System;
```

et

```
Console.WriteLine(Environment.UserName);
```

pour produire le même résultat.

Petit à petit vous allez retenir beaucoup d'instructions et d'espaces de nom du framework .NET mais il est inimaginable de tout retenir. Il existe une documentation répertoriant tout ce qui compose le framework .NET, c'est ce qu'on appelle la MSDN library. Vous pouvez y accéder via le code web suivant :

▷ MSDN library
Code web : 153597

Par exemple, la documentation de la propriété que nous venons d'utiliser est consultable avec le code web suivant :

▷ Doc `UserName`
Code web : 563051

Nous avons désigné la caverne d'Ali Baba par le mot « framework .NET ». Pour être plus précis, la désignation exacte de cette caverne est : « la bibliothèque de classes .NET ». Le mot « framework .NET » est en général utilisé par abus de langage (et vous avez remarqué que j'ai moi-même abusé du langage !) et représente également cette bibliothèque de classes. Nous reviendrons plus tard sur ce qu'est exactement une classe, pour l'instant, vous pouvez voir ça comme des composants. Le framework serait donc une bibliothèque de composants.

Référencer une assembly

Ça y est, nous savons accéder au framework .NET.



Mais d'ailleurs, comment se fait-il que nous puissions accéder aux méthodes du framework .NET sans nous poser de question ? Il est magique ce framework ? Où se trouve le code qui permet de récupérer la date du jour ?

Judicieuse question. Si on y réfléchit, il doit falloir beaucoup de code pour renvoyer la date du jour ! Déjà, il faut aller la lire dans l'horloge système, il faut peut-être l'adapter au fuseau horaire, la formater d'une façon particulière en fonction de la langue, etc.

Tout ça est déjà fait, heureusement, dans la bibliothèque de méthodes du framework .NET.



Alors, où est-elle cette bibliothèque ? Et le reste ?

Dans des assemblys bien sûr ! Comme on l'a vu, les assemblys possèdent des fragments de code compilés en langage intermédiaire. S'ils sont réutilisables, ils se trouvent dans des fichiers dont l'extension est `.dll`.

Le framework .NET est composé d'une multitude d'assemblys qui sont installées sur votre système d'exploitation, dans le « GAC » [2] qui est un endroit où sont stockées ces assemblys afin de les rendre accessibles depuis nos programmes. Pour pouvoir utiliser ces assemblys dans notre programme, nous devons indiquer que nous voulons les utiliser. Pour ça, il faut les référencer dans le projet. Si l'on déplie les références dans notre explorateur de solutions, nous pouvons voir que nous avons déjà pas mal de références qui ont été ajoutées par Visual C# Express (voir figure 9.3).

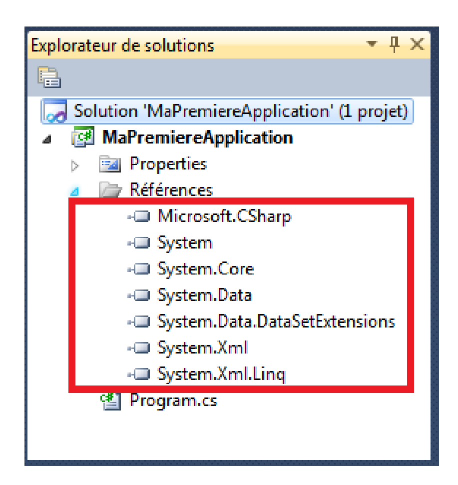


FIGURE 9.3 – Les références de notre application console

Ce sont des assemblys qui sont très souvent utilisées, c'est pour ça que Visual C# Express nous les a automatiquement référencées. Toujours ce souci de nous simplifier le travail. Qu'il est sympa !

Prenons par exemple la référence `System.Xml.` Son nom nous suggère que dedans est réuni tout ce qu'il nous faut pour manipuler le XML. Commençons par taper `System.Xml.`, la complétion automatique nous propose plusieurs choses (voir figure 9.4).

On ne sait pas du tout à quoi elle sert, mais déclarons par exemple une variable de l'énumération `ConformanceLevel` :

2. GAC est l'acronyme de *Global Assembly Cache*.

```
static void Main(string[] args)
{
    System.Xml.
}
```

ConformanceLevel
DtdProcessing
EntityHandling
Formatting
IHasXmlNode
IXmlLineInfo
IXmlNamespaceResolver
Linq
NamespaceHandling

enum System.Xml.ConformanceLevel
Spécifie l'ampleur des contrôles d'entrée ou de sortie que les
System.Xml.XmlWriter créés effectuent.

FIGURE 9.4 – L'espace de nom `System.Xml` est accessible si la référence est présente

```
System.Xml.ConformanceLevel level;
```

et compilons. Pas d'erreur de compilation. Supprimez la référence à `System.Xml.` (clic droit, `Supprimer`) et compilez à nouveau, vous verrez que `ConformanceLevel` est désormais souligné en rouge, signe qu'il y a un problème (voir figure 9.5).



FIGURE 9.5 – Erreur de compilation car la référence a été supprimée

Par ailleurs, la compilation provoque l'erreur suivante :

```
Le type ou le nom d'espace de noms 'ConformanceLevel' n'existe
    pas dans l'espace de noms 'System.Xml' (une référence d'
    assembly est-elle manquante ?)
```

Loin d'être bête, Visual C# Express nous affiche un message d'erreur plutôt explicite.

En effet, cette énumération est introuvable car elle est définie dans une assembly qui n'est pas référencée. C'est comme si on nous demandait de prendre le marteau pour enfoncer un clou, mais que le marteau n'est pas à côté de nous, mais au garage bien rangé ! Il faut donc référencer le marteau afin de pouvoir l'utiliser. Pour ce faire, faisons un clic sur `Références` et ajoutons une référence (voir figure 9.6).

Ici, nous avons plusieurs onglets (voir figure 9.7). Selon la version de Visual Studio que vous possédez, vous aurez peut-être une présentation légèrement différente.

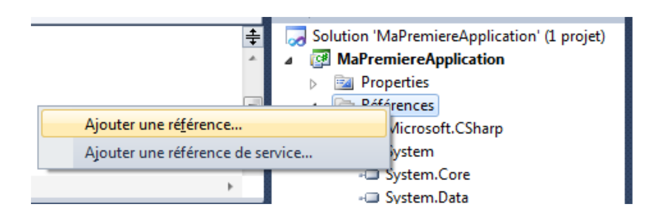


FIGURE 9.6 – Ajouter une référence

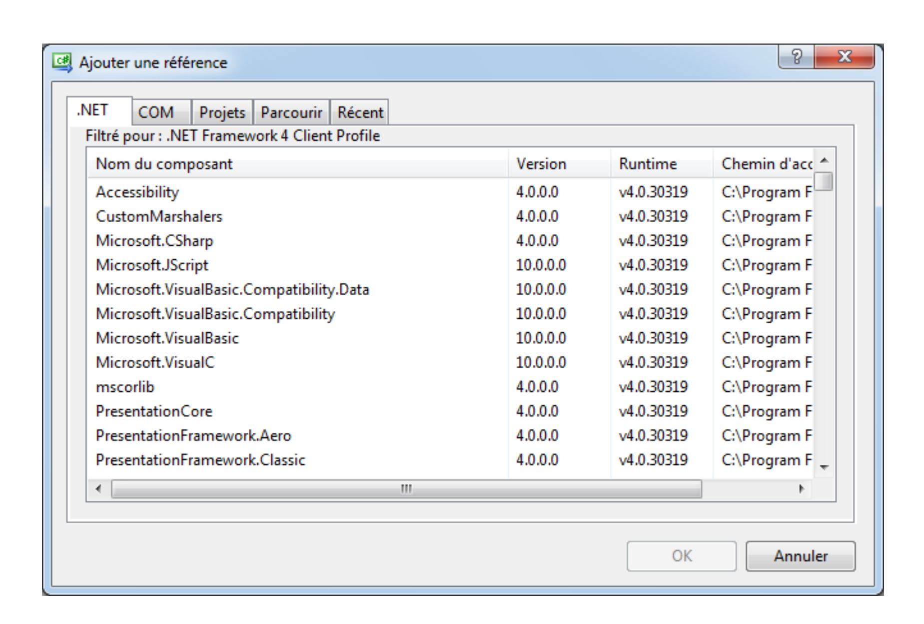


FIGURE 9.7 – Liste des assemblys présentes dans le GAC pouvant être référencées

Chaque onglet permet d'ajouter une référence à une assembly.

- L'onglet `.NET` permet de référencer une assembly présente dans le GAC, c'est ici que nous viendrons chercher les assemblys du framework .NET.
- L'onglet `COM` permet de référencer une dll COM. Vous ne savez pas ce qu'est COM ? On peut dire que c'est l'ancêtre de l'assembly. Techniquement, on ne référence pas directement une dll COM, mais Visual C# Express génère ce qu'on appelle un wrapper permettant d'utiliser la dll COM. Un wrapper est en fait une espèce de traducteur permettant d'utiliser la dll COM comme si c'était une assembly. Mais ne nous attardons pas sur ce point qui ne va pas nous servir, nous qui sommes débutants.
- L'onglet `Projets` permet de référencer des assemblys qui se trouvent dans notre solution. Dans la mesure où notre solution ne contient qu'un seul projet, l'onglet sera vide. Nous verrons plus tard comment utiliser cet onglet lorsque nous ajouterons des assemblys à nos solutions.
- L'onglet `Parcourir` va permettre de référencer une assembly depuis un emplacement sur le disque dur. Cela peut-être une assembly tierce fournie par un collègue, ou par un revendeur, etc.
- Enfin, l'onglet `Récent`, comme son nom le suggère, permet de référencer des assemblys récemment référencées.

Retournons à nos moutons, repartons sur l'onglet `.NET` et recherchons l'assembly que nous avons supprimée, à savoir `System.Xml`.

Une fois trouvée, appuyez sur `OK`. Maintenant que la référence est ajoutée, nous pouvons à nouveau utiliser cette énumération... Ouf !



Je ne comprends pas, j'ai supprimé toutes les références, mais j'arrive quand même à utiliser la date, le nom de l'utilisateur ou la méthode `Console.WriteLine`. Si j'ai bien compris, je ne devrais pas pouvoir utiliser des méthodes dont les assemblys ne sont pas référencées... C'est normal ?

Eh oui, il existe une assembly qui n'apparaît pas dans les références et qui contient tout le cœur du framework .NET. Cette assembly doit obligatoirement être référencée, il s'agit de `mscorlib.dll`. Comme elle est obligatoire, elle est référencée par défaut dès que l'on crée un projet.

D'autres exemples

Il arrivera souvent que vous ayez besoin d'une fonctionnalité trouvée dans la documentation ou sur internet et qu'il faille ajouter une référence. Ce sera peut-être une référence au framework .NET, à des bibliothèques tierces ou à vous. Nous verrons plus loin comment créer nos propres bibliothèques.

Dans la documentation MSDN, il est toujours indiqué quelle assembly il faut référencer pour utiliser une fonctionnalité. Prenons par exemple la propriété `DateTime.Now`, la documentation nous dit :

```
Espace de noms :  System
Assembly :  mscorlib (dans mscorlib.dll)
```

Cela veut donc dire qu'on utilise la date du jour en utilisant l'assembly obligatoire `mscorlib` et la fonctionnalité se trouve dans l'espace de nom `System`, comme déjà vu.

Il n'y a donc rien à faire pour pouvoir utiliser `DateTime.Now`. En imaginant que nous ayons besoin de faire un programme qui utilise des nombres complexes, vous allez sûrement avoir besoin du type `Complex`, trouvé dans la documentation adéquate, via le code web suivant :

▷ Doc `complex`
Code web : 836961

Pour l'utiliser, comme indiqué, il va falloir référencer l'assembly `System.Numerics.dll`. Rien de plus simple, répétons la procédure pour référencer une assembly et allons trouver `System.Numerics.dll`, comme indiqué à la figure 9.8.

FIGURE 9.8 – Référencer l'assembly `System.Numerics`

Ensuite, nous pourrons utiliser par exemple le code suivant :

```
Complex c = Complex.One;
Console.WriteLine(c);
Console.WriteLine("Partie réelle : " + c.Real);
Console.WriteLine("Partie imaginaire : " + c.Imaginary);

Console.WriteLine(Complex.Conjugate(Complex.
    FromPolarCoordinates(1.0, 45 * Math.PI / 180)));
```

Sachant qu'il aura fallu bien sûr ajouter :

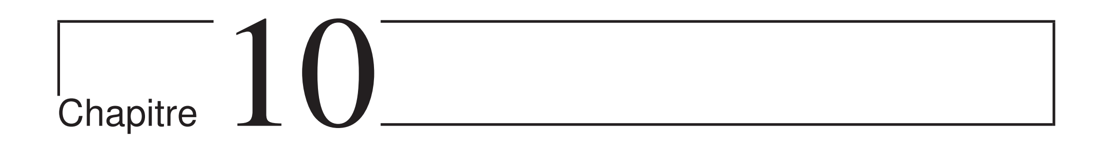
```
using System.Numerics;
```

permettant d'éviter de préfixer `Complex` par `System.Numerics`. Mais ça, vous l'aviez trouvé tout seul, n'est-ce pas ?

```
(1, 0)
Partie réelle : 1
Partie imaginaire : 0
(0,707106781186548, -0,707106781186547)
```

En résumé

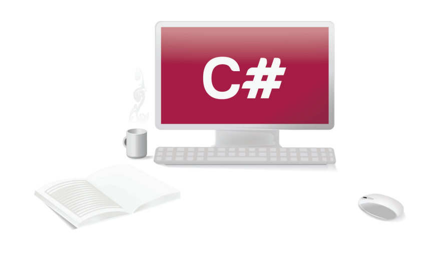
- Le framework .NET est un ensemble d'assemblys qu'il faut référencer dans son projet afin d'avoir accès à leurs fonctionnalités.
- On utilise le mot clé `using` pour inclure un espace de nom comme raccourci dans son programme, ce qui permet de ne pas avoir à préfixer les types de leurs espaces de noms complets.



TP : bonjour c'est le week-end !

Difficulté : 

Bienvenue dans ce premier TP ! Vous avez pu découvrir dans les chapitres précédents les premières bases du langage C# permettant la construction d'applications. Il est grand temps de mettre en pratique ce que nous avons appris. C'est ici l'occasion pour vous de tester vos connaissances et de valider ce que vous appris en réalisant cet exercice.



Instructions pour réaliser le TP

Le but est de créer une petite application qui affiche un message différent en fonction du nom de l'utilisateur et du moment de la journée :

- « Bonjour X » pour la tranche horaire 9h > 18h, les lundis, mardis, mercredis, jeudis et vendredis ;
- « Bonsoir X » pour la tranche horaire 18h > 9h, les lundis, mardis, mercredis, jeudis ;
- « Bon week-end X » pour la tranche horaire vendredi 18h > lundi 9h.

Peut-être cela vous paraît-il simple, dans ce cas, foncez et réalisez cette première application tout seuls. Sinon, décortiquons un peu l'énoncé de ce TP pour éviter d'être perdus !

Pour réaliser ce premier TP, vous allez avoir besoin de plusieurs choses. Dans un premier temps, il faut afficher le nom de l'utilisateur, c'est une chose que nous avons déjà faite en allant puiser dans les fonctionnalités du framework .NET.

Vous aurez besoin également de récupérer l'heure courante pour la comparer aux tranches horaires souhaitées. Vous avez déjà vu comment récupérer la date courante. Pour pouvoir récupérer l'heure de la date courante, il vous faudra utiliser l'instruction `DateTime.Now.Hour` qui renvoie un entier représentant l'heure du jour.

Pour comparer l'heure avec des valeurs entières il vous faudra utiliser les opérateurs de comparaison et les instructions conditionnelles que nous avons vues précédemment.

Pour traiter le cas spécial du jour de la semaine, vous aurez besoin que le framework .NET vous indique quel jour nous sommes. C'est le rôle de l'instruction `DateTime.Now.DayOfWeek` qui est une énumération indiquant le jour de la semaine.Les différentes valeurs sont consultables via le code web suivant :

▷ Doc `DayOfWeek`
Code web : 921325 .

Pour plus de clarté, nous les reprenons ici :

Valeur	Traduction
Sunday	Dimanche
Monday	Lundi
Tuesday	Mardi
Wednesday	Mercredi
Thursday	Jeudi
Friday	Vendredi
Saturday	Samedi

Il ne restera plus qu'à comparer deux valeurs d'énumération, comme on l'a vu dans le chapitre sur les énumérations.

Voilà, vous avez tous les outils en main pour réaliser ce premier TP ! N'hésitez pas à revenir sur les chapitres précédents si vous avez un doute sur la syntaxe ou sur les instructions à réaliser. On ne peut pas apprendre un langage par cœur du premier coup.

À vous de jouer !

Correction

Vous êtes autorisés à lire cette correction uniquement si vous vous êtes arraché les cheveux sur ce TP ! Je vois qu'il vous en reste, encore un effort ! Si vous avez réussi avec brio le TP, vous pourrez également comparer votre travail au mien.

Quoi qu'il en soit, voici la correction que je propose. Bien évidemment, il peut y en avoir plusieurs, mais elle contient les informations nécessaires pour la réalisation de ce TP.

Première chose à faire : créer un projet de type console. J'ai ensuite ajouté le code suivant :

```
static void Main(string[] args)
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Saturday ||
        DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
    {
        // nous sommes le week-end
        AfficherBonWeekEnd();
    }
    else
    {
        // nous sommes en semaine

        if (DateTime.Now.DayOfWeek == DayOfWeek.Monday &&
            DateTime.Now.Hour < 9)
        {
            // nous sommes le lundi matin
            AfficherBonWeekEnd();
        }
        else
        {
            if (DateTime.Now.Hour >= 9 && DateTime.Now.Hour <
                18)
            {
                // nous sommes dans la journée
                AfficherBonjour();
            }
            else
            {
                // nous sommes en soirée

                if (DateTime.Now.DayOfWeek == DayOfWeek.Friday
                    && DateTime.Now.Hour >= 18)
                {
                    // nous sommes le vendredi soir
                    AfficherBonWeekEnd();
```

```
                    }
                    else
                    {
                        AfficherBonsoir();
                    }
                }
            }
        }
}

static void AfficherBonWeekEnd()
{
    Console.WriteLine("Bon week-end " + Environment.UserName);
}

static void AfficherBonjour()
{
    Console.WriteLine("Bonjour " + Environment.UserName);
}

static void AfficherBonsoir()
{
    Console.WriteLine("Bonsoir " + Environment.UserName);
}
```

Détaillons un peu ce code :

Au chargement du programme (méthode `Main`) nous faisons les comparaisons adéquates. Dans un premier temps, nous testons le jour de la semaine de la date courante (`DateTime.Now.DayOfWeek`) et nous la comparons aux valeurs représentant Samedi et Dimanche. Si c'est le cas, alors nous appelons une méthode qui affiche le message « Bon week-end » avec le nom de l'utilisateur courant que nous pouvons récupérer avec `Environment.UserName`.

Si nous ne sommes pas le week-end, alors nous testons l'heure de la date courante avec `DateTime.Now.Hour`. Si nous sommes le lundi matin avant 9h, alors nous continuons à afficher « Bon week-end ». Sinon, si nous sommes dans la tranche horaire 9h – 18h alors nous pouvons appeler la méthode qui affiche « Bonjour ». Dans le cas contraire, il reste juste à vérifier que nous ne sommes pas vendredi soir, qui fait partie du week-end, sinon on peut afficher le message « Bonsoir ».

Et voilà, un bon exercice pour manipuler les conditions et les énumérations. . .

Aller plus loin

Ce TP n'était pas très compliqué, il nous a permis de vérifier que nous avions bien compris le principe des `if` et que nous savions appeler des éléments du framework .NET.

Nous aurions pu simplifier l'écriture de l'application en compliquant en peu les tests avec une combinaison de conditions. Par exemple, on pourrait avoir :

```
if (DateTime.Now.DayOfWeek == DayOfWeek.Saturday ||
    DateTime.Now.DayOfWeek == DayOfWeek.Sunday ||
    (DateTime.Now.DayOfWeek == DayOfWeek.Monday && DateTime.Now
        .Hour < 9) ||
    (DateTime.Now.DayOfWeek == DayOfWeek.Friday && DateTime.Now
        .Hour >= 18))
{
    // nous sommes le week-end
    AfficherBonWeekEnd();
}
else
{
    // nous sommes en semaine
    if (DateTime.Now.Hour >= 9 && DateTime.Now.Hour < 18)
    {
        // nous sommes dans la journée
        AfficherBonjour();
    }
    else
    {
        AfficherBonsoir();
    }
}
```

Le premier test permet de vérifier que nous sommes soit samedi, soit dimanche, soit que nous sommes lundi et que l'heure est inférieure à 9, soit que nous sommes vendredi et que l'heure est supérieure à 18. Nous avons, pour ce faire, combiné les tests avec l'opérateur logique « OU » : ||. Remarquons que les parenthèses nous permettent d'agir sur l'ordre d'évaluation des conditions. Pour que ce soit le week-end, il faut bien sûr être « vendredi et que l'heure soit supérieure à 18 » ou « lundi et que l'heure soit inférieure à 9 » ou samedi ou dimanche.

On pourrait encore simplifier en évitant de solliciter à chaque fois le framework .NET pour obtenir la date courante. Il suffit de stocker la date courante dans une variable de type `DateTime`. Ce qui donnerait :

```
DateTime dateCourante = DateTime.Now;

if (dateCourante.DayOfWeek == DayOfWeek.Saturday ||
    dateCourante.DayOfWeek == DayOfWeek.Sunday ||
    (dateCourante.DayOfWeek == DayOfWeek.Monday && dateCourante
        .Hour < 9) ||
    (dateCourante.DayOfWeek == DayOfWeek.Friday && dateCourante
        .Hour >= 18))
{
    // nous sommes le week-end
    AfficherBonWeekEnd();
}
```

```
else
{
    // nous sommes en semaine
    if (dateCourante.Hour >= 9 && dateCourante.Hour < 18)
    {
        // nous sommes dans la journée
        AfficherBonjour();
    }
    else
    {
        AfficherBonsoir();
    }
}
```

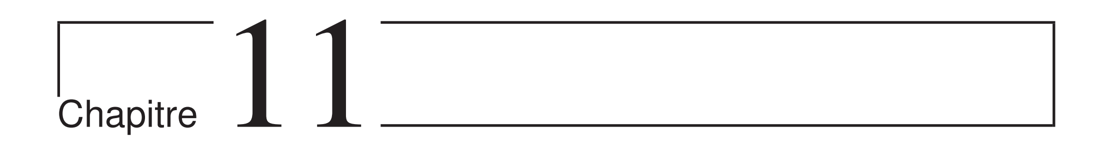
On utilise ici le type `DateTime` comme le type `string` ou `int`. Il sert à gérer les dates et l'heure. Il est légèrement différent des types que nous avons vus pour l'instant, nous ne nous attarderons pas dessus. Nous aurons l'occasion de découvrir de quoi il s'agit dans la partie suivante. Cette optimisation n'a rien d'extraordinaire, mais cela nous évite un appel répété au framework .NET.

Voilà pour ce TP. J'espère que vous aurez réussi avec brio l'exercice. Vous avez pu remarquer que ce n'était pas trop difficile. Il a simplement fallu réfléchir à comment imbriquer correctement nos conditions.

N'hésitez pas à pratiquer et à vous entraîner avec d'autres problèmes de votre cru. Si vous avez la moindre hésitation, vous pouvez relire les chapitres précédents. Vous verrez que nous aurons l'occasion d'énormément utiliser ces instructions conditionnelles dans tous les programmes que nous allons écrire !

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 106683

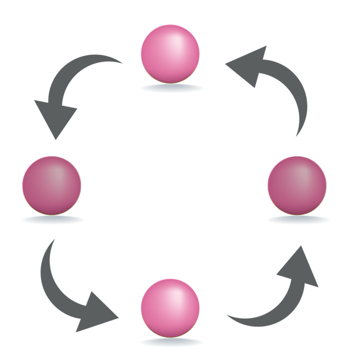


Chapitre 11

Les boucles

Difficulté : 

Nous les avons évoquées rapidement un peu plus tôt en parlant des tableaux et des listes. Dans ce chapitre nous allons décrire plus précisément les boucles.
Elles sont souvent utilisées pour parcourir des éléments énumérables, comme le sont les tableaux ou les listes. Elles peuvent également être utilisées pour effectuer la même action tant qu'une condition n'est pas réalisée.



La boucle `for`

La première instruction que nous avons aperçue est la boucle `for`. Elle permet de répéter un bout de code tant qu'une condition est vraie. Souvent cette condition est un compteur. Nous pouvons par exemple afficher un message 50 fois avec le code suivant :

```
int compteur;
for (compteur = 0; compteur < 50; compteur++)
{
    Console.WriteLine("Bonjour C#");
}
```

Ce qui affichera :

```
Bonjour C#
Bonjour C#
Bonjour C#
[... 50 fois en tout, mais abrégé pour économiser du papier...
   !]
Bonjour C#
Bonjour C#
Bonjour C#
```

Nous définissons ici un entier `compteur`. Il est initialisé à 0 en début de boucle (`compteur = 0`). Après chaque exécution du bloc de code du `for`, c'est-à-dire à chaque itération, il va afficher « Bonjour C# ». À la fin de chaque itération, la variable `compteur` est incrémentée (`compteur++`) et nous recommençons l'opération tant que la condition « compteur < 50 » est vraie.

Bien sûr, la variable `compteur` est accessible dans la boucle et change de valeur à chaque itération.

```
int compteur;
for (compteur = 0; compteur < 50; compteur++)
{
    Console.WriteLine("Bonjour C# " + compteur);
}
```

Le code précédent affichera la valeur de la variable après chaque bonjour, donc de 0 à 49. En effet, quand `compteur` passe à 50, la condition n'est plus vraie et on passe aux instructions suivantes :

```
Bonjour C# 0
Bonjour C# 1
Bonjour C# 2
Bonjour C# 3
[... abrégé également...]
Bonjour C# 48
Bonjour C# 49
```

En général, on utilise la boucle `for` pour parcourir un tableau. Ainsi, nous pourrons utiliser le compteur comme indice pour accéder aux éléments du tableau :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
int indice;
for (indice = 0; indice < 7; indice++)
{
    Console.WriteLine(jours[indice]);
}
```

Dans cette boucle, vu qu'à la première itération `indice` vaut 0, nous afficherons l'élément du tableau à la position 0, à savoir « Lundi ». À l'itération suivante, `indice` passe à 1, nous affichons l'élément du tableau à la position 1, à savoir « Mardi », et ainsi de suite :

```
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
```

Attention à ce que l'indice ne dépasse pas la taille du tableau, sinon l'accès à un indice en dehors des limites du tableau provoquera une erreur à l'exécution. Pour éviter ceci, on utilise en général la taille du tableau comme condition de fin :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
int indice;
for (indice = 0; indice < jours.Length; indice++)
{
    Console.WriteLine(jours[indice]);
}
```

Ici `jours.Length` renvoie la taille du tableau, à savoir 7.

Il est très courant de boucler sur un tableau en passant tous les éléments un par un, mais il est possible de changer les conditions de départ, les conditions de fin, et l'élément qui influe sur la condition de fin.

Ainsi, l'exemple suivant :

```
int[] chiffres = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (int i = 9; i > 0; i -= 2)
{
    Console.WriteLine(chiffres[i]);
}
```

Permet de parcourir le tableau de 2 en 2 en commençant par la fin :

```
9
7
5
3
1
```

Vous avez pu voir que dans cet exemple, nous avons défini l'entier `i` directement dans l'instruction `for`. C'est une commodité qui permet que la variable `i` n'existe qu'à l'intérieur de la boucle, car sa portée correspond aux accolades qui délimitent le `for`.

Attention, il est possible de faire un peu tout et n'importe quoi dans ces boucles. Aussi il peut arriver que l'on se retrouve avec des bugs, comme des boucles infinies.

Par exemple, le code suivant :

```
for (int indice = 0; indice < 7; indice--)
{
    Console.WriteLine("Test" + indice);
}
```

est une boucle infinie. En effet, on modifie la variable `indice` en la décrémentant. Sauf que la condition de sortie de la boucle est valable pour un indice qui dépasse ou égale la valeur 7, ce qui n'arrivera jamais.

Si on exécute l'application avec ce code, la console va afficher à l'infini le mot « Test » avec son indice. La seule solution pour quitter le programme sera de fermer brutalement l'application.

L'autre solution est d'attendre que le programme se termine...



Mais tu viens de dire que la boucle était infinie ?

Oui c'est vrai, mais en fait, ici on se heurte à un cas limite du C#. C'est à cause de la variable `indice`, qui est un entier que l'on décrémente. Au début il vaut zéro, puis -1, puis -2, etc. Lorsque la variable `indice` arrive à la limite inférieure que le type `int` est capable de gérer, c'est-à-dire -2147483648, se produit alors ce qu'on appelle un **dépassement de capacité**. Sans rentrer dans les détails, il ne peut pas stocker un entier plus petit et donc il boucle et repart à l'entier le plus grand, c'est-à-dire 2147483647.

Donc pour résumer, l'indice fait :

- 0
- -1
- -2
- ...
- -2147483647
- -2147483648

– +2147483647

Et comme là, il se retrouve supérieur à 7, la boucle se termine !

```
Test 0
Test -1
Test -2
[... abrégé ...]
Test -2147483647
Test -2147483648
```

Donc, techniquement, ce n'est pas une boucle infinie, mais on a attendu tellement longtemps pour atteindre ce cas limite que c'est tout comme.

Et surtout, nous tombons sur un cas imprévu. Ici, ça se termine plutôt bien, mais ça aurait pu finir en crash de l'application.



Dans tous les cas, il faut absolument maîtriser ses conditions de sortie de boucle pour éviter la boucle infinie, un des cauchemars du développeur !

La boucle `foreach`

Comme il est très courant d'utiliser les boucles pour parcourir un tableau ou une liste, le C# dispose d'un opérateur particulier : `foreach` que l'on peut traduire par « pour chaque » ; pour chaque élément du tableau faire ceci...

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
foreach (string jour in jours)
{
    Console.WriteLine(jour);
}
```

Cette boucle va nous permettre de parcourir tous les éléments du tableau `jours`. À chaque itération, la boucle va créer une chaîne de caractères `jour` qui contiendra l'élément courant du tableau. À noter que la variable `jour` aura une portée égale au bloc `foreach`. Nous pourrons ainsi utiliser cette valeur dans le corps de la boucle et pourquoi pas l'afficher comme dans l'exemple précédent :

```
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
```

L'instruction `foreach` fonctionne aussi avec les listes. Par exemple le code suivant :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
foreach (string jour in jours)
{
    Console.WriteLine(jour);
}
```

nous permettra d'afficher tous les jours de la semaine contenus dans la liste des jours.

Attention, la boucle `foreach` est une boucle en lecture seule. Cela veut dire qu'il n'est pas possible de modifier l'élément de l'itération en cours.

Par exemple, le code suivant :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
foreach (string jour in jours)
{
    jour = "pas de jour !";
}
```

provoquera l'erreur de compilation suivante :

```
Impossible d'assigner à 'jour', car il s'agit d'un 'variable d'
   itération foreach'
```

Notez d'ailleurs que l'équipe de traduction de Visual C# Express a quelques progrès à faire. . . !

Si nous souhaitons utiliser une boucle pour changer la valeur de notre liste ou de notre tableau, il faudra passer par une boucle `for` :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
for (int i = 0; i < jours.Count; i++ )
{
    jours[i] = "pas de jour !";
}
```

Il vous arrivera aussi sûrement un jour (ça arrive à tous les développeurs) de vouloir modifier une liste en elle-même lors d'une boucle `foreach`. C'est-à-dire lui rajouter un élément, en supprimer un autre, etc. C'est également impossible, car à partir du moment où l'on rentre dans la boucle `foreach`, la liste devient non-modifiable.

Prenons l'exemple, ô combien classique, de la recherche d'une valeur dans une liste pour la supprimer. Nous serions tentés de faire :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
foreach (string jour in jours)
{
```

```
    if (jour == "Mercredi")
        jours.Remove(jour);
}
```

ce qui semblerait tout à fait correct et en plus, ne provoque pas d'erreur de compilation. Sauf que si vous exécutez l'application, vous aurez un message d'erreur :

```
Exception non gérée : System.InvalidOperationException: La
    collection a été modifiée ; l'opération d'énumération peut ne
     pas s'exécuter.
à System.ThrowHelper.ThrowInvalidOperationException(
    ExceptionResource resource)
à System.Collections.Generic.List`1.Enumerator.MoveNextRare()
à System.Collections.Generic.List`1.Enumerator.MoveNext()
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    14
```

Le programme nous indique alors que « la collection a été modifiée » et que « l'opération d'énumération peut ne pas s'exécuter ».

Il est donc impossible de faire notre suppression ainsi.



Comment tu ferais toi ?

Eh bien, plusieurs solutions existent. Celle qui vient en premier à l'esprit est d'utiliser une boucle `for` par exemple :

```
for (int i = 0 ; i < jours.Count ; i++)
{
    if (jours[i]== "Mercredi")
        jours.Remove(jours[i]);
}
```

Cette solution est intéressante ici, mais elle peut poser un problème dans d'autres situations. En effet, vu que nous supprimons un élément de la liste, nous allons nous retrouver avec une incohérence entre l'indice en cours et l'élément que nous essayons d'atteindre. En effet, lorsque le jour courant est mercredi, l'indice `i` vaut 2. Si l'on supprime cet élément, c'est jeudi qui va se retrouver en position 2. Et nous allons rater son analyse car la boucle va continuer à l'indice 3, qui sera vendredi. On peut éviter ce problème en parcourant la boucle à l'envers :

```
for (int i = jours.Count - 1; i >= 0; i--)
{
    if (jours[i] == "Mercredi")
        jours.Remove(jours[i]);
}
```



D'une manière générale, il ne faut pas modifier les collections que nous sommes en train de parcourir.

Nous n'étudierons pas les autres solutions car elles font appel à des notions que nous verrons en détail plus tard.

Après avoir lu ceci, vous pourriez avoir l'impression que la boucle `foreach` n'est pas souple et difficilement exploitable : autant utiliser autre chose... Vous verrez à l'utilisation que non, elle est en fait très pratique. Il faut simplement connaître ses limites. Voilà qui est chose faite !

Nous avons vu que l'instruction `foreach` permettait de boucler sur tous les éléments d'un tableau ou d'une liste. En fait, il est possible de parcourir tous les types qui sont **énumérables**. Nous verrons plus loin ce qui caractérise un type énumérable, car pour l'instant, c'est un secret ! Chuuut...

La boucle `while`

La boucle `while` est en général moins utilisée que `for` ou `foreach`. C'est la boucle qui va nous permettre de faire quelque chose tant qu'une condition n'est pas vérifiée. Elle ressemble de loin à la boucle `for`, mais la boucle `for` se spécialise dans le parcours de tableau tandis que la boucle `while` est plus générique.

Par exemple :

```
int i = 0;
while (i < 50)
{
    Console.WriteLine("Bonjour C#");
    i++;
}
```

Ce code permet d'écrire « Bonjour C# » 50 fois de suite. Ici, la condition du `while` est évaluée en début de boucle.

Dans l'exemple suivant :

```
int i = 0;
do
{
    Console.WriteLine("Bonjour C#");
    i++;
}
while (i < 50);
```

La condition de sortie de boucle est évaluée à la fin de la boucle. L'utilisation du mot-clé `do` permet d'indiquer le début de la boucle.

Concrètement cela veut dire que dans le premier exemple, le code à l'intérieur de la boucle peut ne jamais être exécuté, si par exemple l'entier `i` est initialisé à 50. *A*

contrario, on passera au moins une fois dans le corps de la boucle du second exemple, même si l'entier `i` est initialisé à 50 car la condition de sortie de boucle est évaluée à la fin.

Les conditions de sortie de boucles ne portent pas forcément sur un compteur ou un indice, n'importe quelle expression peut être évaluée. Par exemple :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
int i = 0;
bool trouve = false;
while (!trouve)
{
    string valeur = jours[i];
    if (valeur == "Jeudi")
    {
        trouve = true;
    }
    else
    {
        i++;
    }
}
Console.WriteLine("Trouvé à l'indice " + i);
```

Le code précédent va répéter les instructions contenues dans la boucle `while` tant que le booléen `trouve` sera faux (c'est-à-dire qu'on va s'arrêter dès que le booléen sera vrai). Nous analysons la valeur pour l'indice `i`. Si la valeur est celle cherchée, alors nous passons le booléen à vrai et nous pourrons sortir de la boucle. Sinon, nous incrémentons l'indice pour passer au suivant.

Attention encore aux valeurs de sortie de la boucle. Si nous ne trouvons pas la chaîne recherchée, alors `i` continuera à s'incrémenter ; le booléen ne passera jamais à vrai, nous resterons bloqués dans la boucle et nous risquons d'atteindre un indice qui n'existe pas dans le tableau. Il serait plus prudent que la condition porte également sur la taille du tableau, par exemple :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
int i = 0;
bool trouve = false;
while (i < jours.Length && !trouve)
{
    string valeur = jours[i];
    if (valeur == "Jeudi")
    {
        trouve = true;
    }
    else
    {
        i++;
```

```
    }
}
if (!trouve)
    Console.WriteLine("Valeur non trouvée");
else
    Console.WriteLine("Trouvé à l'indice " + i);
```

Ainsi, si l'indice est supérieur à la taille du tableau, nous sortons de la boucle et nous éliminons le risque de boucle infinie.

Une erreur classique est que la condition ne devienne jamais vraie à cause d'une erreur de programmation. Par exemple, si j'oublie d'incrémenter la variable `i`, alors à chaque passage de la boucle j'analyserai toujours la première valeur du tableau et je n'atteindrai jamais la taille maximale du tableau ; condition qui me permettrait de sortir de la boucle.



Je ne le répéterai jamais assez : faites bien attention aux conditions de sortie d'une boucle !

Les instructions `break` et `continue`

Il est possible de sortir prématurément d'une boucle grâce à l'instruction `break`. Dès qu'elle est rencontrée, elle sort du bloc de code de la boucle. L'exécution du programme continue alors avec les instructions situées après la boucle. Par exemple :

```
int i = 0;
while (true)
{
    if (i >= 50)
    {
        break;
    }
    Console.WriteLine("Bonjour C#");
    i++;
}
```

Le code précédent pourrait potentiellement produire une boucle infinie. En effet, la condition de sortie du `while` est toujours vraie. Mais l'utilisation du mot-clé `break` nous permettra de sortir de la boucle dès que `i` atteindra la valeur 50. Certes ici, il suffirait que la condition de sortie porte sur l'évaluation de l'entier `i`. Mais il y a des cas où il pourra être judicieux d'utiliser un `break` (surtout lors d'un déménagement !).

C'est le cas pour l'exemple suivant. Imaginons que nous voulions vérifier la présence d'une valeur dans une liste. Pour la trouver, on peut parcourir les éléments de la liste et une fois trouvée, on peut s'arrêter. En effet, il serait inutile de continuer à parcourir le reste des éléments :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
bool trouve = false;
foreach (string jour in jours)
{
    if (jour == "Jeudi")
    {
        trouve = true;
        break;
    }
}
```

Nous nous dispensons ici d'analyser les 3 derniers éléments de la liste.

Il est également possible de passer à l'itération suivante d'une boucle grâce à l'utilisation du mot-clé `continue`. Prenons l'exemple suivant :

```
for (int i = 0; i < 20; i++)
{
    if (i % 2 == 0)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Ici, l'opérateur % est appelé « **modulo** ». Il permet d'obtenir le reste de la division. L'opération `i` % 2 renverra donc 0 quand `i` est pair. Ainsi, dès qu'un nombre pair est trouvé, nous passons à l'itération suivante grâce au mot-clé `continue`. Ce qui fait que nous n'afficherons que les nombres impairs :

```
1
3
5
7
9
11
13
15
17
19
```

Bien sûr, il aurait été possible d'inverser la condition du `if` pour n'exécuter le `Console.WriteLine()` que dans le cas où `i` % 2!= 0. À vous de choisir l'écriture que vous préférez en fonction des cas que vous rencontrez.

En résumé

- On utilise la boucle `for` pour répéter des instructions tant qu'une condition n'est pas vérifiée, les éléments de la condition changeant à chaque itération.

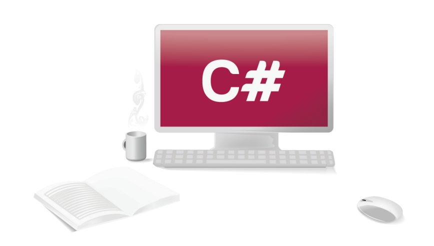
- On utilise en général la boucle `for` pour parcourir un tableau, avec un indice qui s'incrémente à chaque itération.
- La boucle `foreach` est utilisée pour simplifier le parcours des tableaux ou des listes.
- La boucle `while` permet de répéter des instructions tant qu'une condition n'est pas vérifiée. C'est la boucle la plus souple.
- Il faut faire attention aux conditions de sortie d'une boucle afin d'éviter les boucles infinies qui font planter l'application.

Chapitre 12

TP : calculs en boucle

Difficulté : 

Ça y est, grâce au chapitre précédent, vous devez avoir une bonne idée de ce que sont les boucles. Par contre, vous ne voyez peut-être pas encore complètement qu'elles vont vous servir tout le temps. C'est un élément qu'il est primordial de maîtriser. Il vous faut donc vous entraîner pour être bien sûrs d'avoir tout compris. C'est justement l'objectif de ce deuxième TP. Finie la théorie des boucles, place à la pratique en boucle ! Notre but est de réaliser des calculs qui vont avoir besoin des `for` et des `foreach`. Vous êtes prêts ? Alors, c'est parti !



Instructions pour réaliser le TP

Le but de ce TP va être de créer trois méthodes.

La première va servir à calculer la somme d'entiers consécutifs. Si par exemple je veux calculer la somme des entiers de 1 à 10, c'est-à-dire 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10, je vais appeler cette méthode en lui passant en paramètres 1 et 10, c'est-à-dire les bornes des entiers dont il faut faire la somme.

Quelque chose du genre :

```
Console.WriteLine(CalculSommeEntiers(1, 10));
Console.WriteLine(CalculSommeEntiers(1, 100));
```

Sachant que le premier résultat de cet exemple vaut 55 (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55) et le deuxième 5050.

La deuxième méthode acceptera une liste de `double` en paramètres et devra renvoyer la moyenne des `doubles` de la liste. Par exemple :

```
List<double> liste = new List<double> { 1.0, 5.5, 9.9, 2.8, 9.6
    };
Console.WriteLine(CalculMoyenne(liste));
```

Le résultat de cet exemple vaut 5.76.

Enfin, la dernière méthode devra dans un premier temps construire une liste d'entiers de 1 à 100 qui sont des multiples de 3 (3, 6, 9, 12,...). Dans un second temps, construire une autre liste d'entiers de 1 à 100 qui sont des multiples de 5 (5, 10, 15, 20,...). Et dans un dernier temps, il faudra calculer la somme des entiers qui sont communs aux deux listes... vous devez bien sûr trouver 315 comme résultat !

Voilà, c'est à vous de jouer...

Bon, allez, je vais quand même vous donner quelques conseils pour démarrer. Vous n'êtes pas obligés de les lire si vous vous sentez capables de réaliser cet exercice tout seul.

Vous l'aurez évidemment compris, il va falloir utiliser des boucles. Je ne donnerai pas de conseils pour la première méthode, c'est juste pour vous échauffer !

Pour la deuxième méthode, vous allez avoir besoin de diviser la somme de tous les `double` par la taille de la liste. Vous ne savez sans doute pas comment obtenir cette taille. Le principe est le même que pour la taille d'un tableau et vous l'aurez sans doute trouvé si vous fouillez un peu dans les méthodes de la liste. Toujours est-il que pour obtenir la taille d'une liste, on va utiliser `liste.Count`, avec par exemple :

```
int taille = liste.Count;
```

Enfin, pour la dernière méthode, vous allez devoir trouver tous les multiples de 3 et de 5. Le plus simple, à mon avis, pour calculer tous les multiples de 3, est de faire une boucle qui démarre à 3 et d'avancer de 3 en 3 jusqu'à la valeur souhaitée. On fait de même pour les multiples de 5.

Ensuite, il sera nécessaire de faire deux boucles imbriquées afin de déterminer les intersections. C'est-à-dire parcourir la liste des multiples de 3 et à l'intérieur de cette boucle, parcourir la liste des multiples de 5. On compare les deux éléments ; s'ils sont égaux, c'est qu'ils sont communs aux deux listes.

Voilà, vous devriez avoir tous les éléments en main pour réussir ce TP, c'est à vous de jouer !

Correction

J'imagine que vous avez réussi ce TP, non pas sans difficultés, mais à force de tâtonnements, vous avez sans doute réussi. Bravo ! Quoi qu'il en soit, voici la correction que je propose.

– Première méthode :

```
static int CalculSommeEntiers(int borneMin, int borneMax)
{
    int resulat = 0;
    for (int i = borneMin; i <= borneMax ; i++)
    {
        resulat += i;
    }
    return resulat;
}
```

Ce n'était pas très compliqué, il faut dans un premier temps construire une méthode qui renvoie un entier et qui accepte deux entiers en paramètres. Ensuite, on boucle grâce à l'instruction `for` de la borne inférieure à la borne supérieure (incluse, donc il faut utiliser l'opérateur de comparaison $<=$) en incrémentant un compteur de 1 à chaque itération. Nous ajoutons la valeur du compteur à un résultat, que nous retournons en fin de boucle.

– Seconde méthode :

```
static double CalculMoyenne(List<double> liste)
{
    double somme = 0;
    foreach (double valeur in liste)
    {
        somme += valeur;
    }
    return somme / liste.Count;
}
```

Ici, le principe est grosso modo le même, la différence est que la méthode retourne un `double` et accepte une liste de `double` en paramètres. Ici, nous utilisons la boucle `foreach` pour parcourir tous les éléments que nous ajoutons à un résultat. Enfin, nous retournons ce résultat divisé par la taille de la liste.

– Troisième méthode :

```
static int CalculSommeIntersection()
{
    List<int> multiplesDe3 = new List<int>();
    List<int> multiplesDe5 = new List<int>();

    for (int i = 3; i <= 100; i += 3)
    {
        multiplesDe3.Add(i);
    }
    for (int i = 5; i <= 100; i += 5)
    {
        multiplesDe5.Add(i);
    }

    int somme = 0;
    foreach (int m3 in multiplesDe3)
    {
        foreach (int m5 in multiplesDe5)
        {
            if (m3 == m5)
                somme += m3;
        }
    }
    return somme;
}
```

Cette troisième méthode est peut-être la plus compliquée. . .

On commence par créer nos deux listes de multiples. Comme je vous avais conseillé, j'utilise une boucle `for` qui commence à 3 avec un incrément de 3. Comme ça, je suis sûr d'avoir tous les multiples de 3 dans ma liste. C'est le même principe pour les multiples de 5, sachant que dans les deux cas, la condition de sortie est quand l'indice est supérieur à 100.

Ensuite, j'ai mes deux boucles imbriquées où je compare les deux valeurs et si elles sont égales, je rajoute la valeur à la somme globale que je renvoie en fin de méthode. Pour bien comprendre ce qu'il se passe dans les boucles imbriquées, il faut comprendre que nous allons parcourir une unique fois la liste `multiplesDe3` mais que nous allons parcourir autant de fois la liste `multipleDe5` qu'il y a d'éléments dans la liste `multipleDe3`, c'est-à-dire 33 fois. Ce n'est sans doute pas facile de le concevoir dès le début, mais pour vous aider, vous pouvez essayer de vous faire l'algorithme dans la tête :

– on rentre dans la boucle qui parcoure la liste `multiplesDe3` ;
– `m3` vaut 3 ;
– on rentre dans la boucle qui parcoure la liste `multiplesDe5` ;
– `m5` vaut 5 ;
– on compare 3 à 5, ils sont différents ;
– on passe à l'itération suivante de la liste `multiplesDe5` ;

- m5 vaut 10 ;
- on compare 3 à 10, ils sont différents ;
- de même jusqu'à ce qu'on ait fini de parcourir la liste des multiplesDe5 ;
- on passe à l'itération suivante de la liste multiplesDe3 ;
- m3 vaut 6 ;
- on rentre dans la boucle qui parcoure la liste multiplesDe5 ;
- m5 vaut 5 ;
- on compare 6 à 5, ils sont différents ;
- on passe à l'itération suivante de la liste multiplesDe5 ;
- etc.

Aller plus loin

Vous avez remarqué que dans la deuxième méthode, j'utilise une boucle `foreach` pour parcourir la liste :

```
static double CalculMoyenne(List<double> liste)
{
    double somme = 0;
    foreach (double valeur in liste)
    {
        somme += valeur;
    }
    return somme / liste.Count;
}
```

Il est aussi possible de parcourir les listes avec un `for` et d'accéder aux éléments de la liste avec un indice, comme on le fait pour un tableau. Ce qui donnerait :

```
static double CalculMoyenne(List<double> liste)
{
    double somme = 0;
    for (int i = 0; i < liste.Count; i++)
    {
        somme += liste[i];
    }
    return somme / liste.Count;
}
```

Notez qu'on se sert de `liste.Count` pour obtenir la taille de la liste et qu'on accède à l'élément courant avec `liste[i]`. Je reconnais que la boucle `foreach` est plus explicite, mais cela peut être utile de connaître le parcours d'une liste avec la boucle `for`. À vous de voir.

Vous aurez également noté ma technique pour avoir des multiples de 3 et de 5. Il y en a plein d'autres. Je pense à une en particulier et qui fait appel à des notions que nous avons vues auparavant : la division entière et la division de `double`.

Pour savoir si un nombre est un multiple d'un autre, il suffit de les diviser entre eux et de voir s'il y a un reste à la division. On peut faire cela de deux façons. La première, en comparant la division entière avec la division `double` et en vérifiant que le résultat est le même. Si le résultat est le même, c'est qu'il n'y a pas de chiffres après la virgule :

```
for (int i = 1; i <= 100; i++)
{
    if (i / 3 == i / 3.0)
        multiplesDe3.Add(i);
    if (i / 5 == i / 5.0)
        multiplesDe5.Add(i);
}
```



Cette technique fait appel à des notions que nous n'avons pas encore vues : comment se fait-il qu'on puisse comparer un entier à un `double` ? Nous le verrons un peu plus tard.

L'autre solution est d'utiliser l'opérateur modulo que nous avons vu précédemment qui fait justement ça :

```
for (int i = 1; i <= 100; i++)
{
    if (i % 3 == 0)
        multiplesDe3.Add(i);
    if (i % 5 == 0)
        multiplesDe5.Add(i);
}
```

L'avantage de ces deux techniques est qu'on peut construire les deux listes de multiples en une seule boucle.

Voilà, c'est fini pour ce TP. N'hésitez pas à vous faire la main sur ces boucles car il est fondamental que vous les maîtrisiez. Faites-vous plaisir, tentez de les repousser dans leurs limites, essayez de résoudre d'autres problèmes... L'important, c'est de pratiquer.

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 764808

Deuxième partie

Un peu plus loin avec le C#

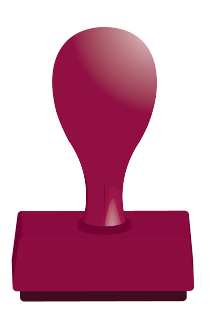
Chapitre 13

Les conversions entre les types

Difficulté : 

Nous avons vu que le C# est un langage qui possède plein de types de données différents : entier, décimal, chaîne de caractères, etc.

Dans nos programmes, nous allons très souvent avoir besoin de manipuler des données entre elles alors qu'elles ne sont pas forcément du même type. Lorsque cela sera possible, nous aurons besoin de convertir un type de données en un autre.



Entre les types compatibles : le casting

C'est l'heure de faire la sélection des types les plus performants, place au casting ! Le jury est en place, à vos SMS pour voter.

Ah, on me fait signe qu'il ne s'agirait pas de ce casting-là…



Le casting est simplement l'action de convertir la valeur d'un type dans un autre.

Plus précisément, cela fonctionne pour les types qui sont compatibles entre eux, entendez par là « qui se ressemblent ».

Par exemple, l'entier et le petit entier se ressemblent. Pour rappel, nous avons :

Type	Description
short	Entier de -32768 à 32767
int	Entier de -2147483648 à 2147483647

Ils ne diffèrent que par leur capacité. Le `short` ne pourra pas contenir le nombre 100000 par exemple, alors que l'`int` le pourra.

Nous ne l'avons pas encore fait, mais le C# nous autorise à faire :

```
short s = 200;
int i = s;
```

En effet, il est toujours possible de stocker un petit entier dans un grand. Peu importe la valeur de `s`, `i` sera toujours à même de contenir sa valeur. C'est comme dans une voiture. Si nous arrivons à tenir à 5 dans un monospace, nous pourrons facilement tenir à 5 dans un bus.

L'inverse n'est pas vrai bien sûr. Si nous tenons à 100 dans un bus, ce n'est pas certain que nous pourrons tenir dans le monospace, même en se serrant bien.

Ce qui fait que si nous faisons :

```
int i = 100000;
short s = i;
```

nous aurons l'erreur de compilation suivante :

```
Impossible de convertir implicitement le type 'int' en 'short'.
   Une conversion explicite existe (un cast est-il manquant ?)
```

Eh oui, comment pouvons-nous faire rentrer 100000 dans un type qui ne peut aller que jusqu'à 32767 ? Le compilateur le sait bien.

Vous remarquerez que nous aurons cependant la même erreur si nous tentons de faire :

```
int i = 200;
short s = i;
```



Mais pourquoi ? Le `short` est bien capable de stocker la valeur 200 ?

Oui tout à fait, mais le compilateur nous avertit quand même. Il nous dit :



« Attention, vous essayez de faire rentrer les personnes du bus dans un monospace, êtes-vous bien sûr ? »

Nous avons envie de lui répondre :



« Oui, oui, je sais qu'il y a très peu de personnes dans le bus et qu'ils pourront rentrer sans aucun problème dans le monospace. Fais-moi confiance ! »

Eh bien, ceci s'écrit en C# de la manière suivante :

```
int i = 200;
short s = (short)i;
```



Nous utilisons ce qu'on appelle un cast explicite.

Pour faire un tel cast, il suffit de faire précéder la variable à convertir du nom du type souhaité entre parenthèses. Cela permet d'indiquer à notre compilateur que nous savons ce que nous faisons, et que l'entier tiendra correctement dans le petit entier.

Mais attention, le compilateur nous fait confiance. Nous sommes le boss ! Cela implique une certaine responsabilité, il ne faut pas faire n'importe quoi.

Si nous faisons :

```
int i = 40000;
short s = (short)i;
Console.WriteLine(s);
Console.WriteLine(i);
```

nous tentons de faire rentrer un trop gros entier dans ce qu'est capable de stocker le petit entier.

Si nous exécutons notre application, nous aurons un résultat surprenant :

```
-25536
40000
```

Le résultat n'est pas du tout celui attendu. Nous avons fait n'importe quoi, le C# nous a punis !

En fait, plus précisément, il s'est passé ce qu'on appelle un dépassement de capacité. Nous l'avons déjà vu lors du chapitre sur les boucles. Ici, il s'est produit la même chose. Le petit entier est allé à sa valeur maximale puis a bouclé en repartant de sa valeur minimale.

Bref, tout ça pour dire que nous obtenons un résultat inattendu. Il faut donc faire attention à ce que l'on fait et honorer la confiance que nous porte le compilateur en faisant bien attention à ce que l'on caste. [1].

D'une façon similaire à l'entier et au petit entier, l'entier et le double se ressemblent un peu. Ce sont tous les deux des types permettant de contenir des nombres. Le premier permet de contenir des nombres entiers, le deuxième peut contenir des nombres à virgules.

Ainsi, nous pouvons écrire le code C# suivant :

```
int i = 20;
double d = i;
```

En effet, un double est plus précis qu'un entier. Il est capable d'avoir des chiffres après la virgule alors que l'entier ne le peut pas. Ce qui fait que le double est entièrement capable de stocker toute la valeur d'un entier sans perdre d'information dans cette affectation.

Par contre, comme on peut s'y attendre, l'inverse n'est pas possible. Le code suivant :

```
double prix = 125.55;
int valeur = prix;
```

produira l'erreur de compilation suivante :

```
Impossible de convertir implicitement le type 'double' en 'int'.
    Une conversion explicite existe (un cast est-il manquant ?)
```

En effet, un double étant plus précis qu'un entier, si nous mettons ce double dans l'entier nous allons perdre notamment les chiffres après la virgule.

Il restera encore une fois à demander au compilateur de nous faire confiance : « OK, ceci est un double, mais ce n'est pas grave, je veux l'utiliser comme un entier ! Oui, oui, même si je sais que je vais perdre les chiffres après la virgule ».

Ce qui s'écrit en C# de cette manière, comme nous l'avons vu :

```
double prix = 125.55;
int valeur = (int)prix; // valeur vaut 125
```

Nous faisons précéder la variable `prix` du type dans lequel nous voulons la convertir en utilisant les parenthèses.

1. Nous utiliserons souvent cet anglicisme, qui signifiera bien sûr que nous convertissons des types qui se ressemblent entre eux, pour le plus grand malheur des professeurs de français.



En l'occurrence, on peut se servir de ce cast pour récupérer la partie entière d'un nombre à virgule.

C'est plutôt sympa comme instruction. Mais n'oubliez pas que cette opération est possible uniquement avec les types qui se ressemblent entre eux.

Par exemple, le cast suivant est impossible :

```
string nombre = "123";
int valeur = (int)nombre;
```

car les deux types sont trop différents et sont incompatibles entre eux. Même si la chaîne de caractères représente un nombre.

Nous verrons plus loin comment convertir une chaîne en entier. Pour l'instant, Visual C# Express nous génère une erreur de compilation :

```
Impossible de convertir le type 'string' en 'int'
```



Le message d'erreur est plutôt explicite. Il ne nous propose même pas d'utiliser de cast, il considère que les types sont vraiment trop différents !

Nous avons vu précédemment que les énumérations représentaient des valeurs entières. Il est donc possible de récupérer l'entier correspondant à une valeur de l'énumération grâce à un cast.

Par exemple, en considérant l'énumération suivante :

```
enum Jours
{
    Lundi = 5, // lundi vaut 5
    Mardi, // mardi vaut 6
    Mercredi = 9, // mercredi vaut 9
    Jeudi = 10, // jeudi vaut 10
    Vendredi, // vendredi vaut 11
    Samedi, // samedi vaut 12
    Dimanche = 20 // dimanche vaut 20
}
```

Le code suivant :

```
int valeur = (int)Jours.Lundi; // valeur vaut 5
```

convertit l'énumération en entier.

Nous verrons que le cast est beaucoup plus puissant que ça, mais pour l'instant, nous n'avons pas assez de connaissances pour tout étudier. Nous y reviendrons dans la partie suivante.

Entre les types incompatibles

Nous avons vu qu'il était facile de convertir des types qui se ressemblent grâce au cast. Il est parfois possible de convertir des types qui ne se ressemblent pas mais qui ont le même sens.

Par exemple, il est possible de convertir une chaîne de caractères qui contient uniquement des chiffres en un nombre (entier, décimal, etc.). Cette conversion va nous servir énormément car dès qu'un utilisateur va saisir une information par le clavier, elle sera représentée par une chaîne de caractères. Donc si on lui demande de saisir un nombre, il faut être capable d'utiliser sa saisie comme un nombre afin de le transformer, de le stocker, etc.

Pour ce faire, il existe plusieurs solutions. La plus simple est d'utiliser la méthode `Convert.ToInt32()` disponible dans le framework .NET. Par exemple :

```
string chaineAge = "20";
int age = Convert.ToInt32(chaineAge); // age vaut 20
```

Cette méthode, bien que simple d'utilisation, présente un inconvénient dans le cas où la chaîne ne représente pas un entier. À ce moment-là, la conversion va échouer et le C# va renvoyer une erreur au moment de l'exécution.

Par exemple :

```
string chaineAge = "vingt ans";
int age = Convert.ToInt32(chaineAge);
```

Si vous exécutez ce bout de code, vous verrez que la console nous affiche ce que l'on appelle une exception :

```
Exception non gérée : System.FormatException: Le format de la
    chaîne d'entrée est incorrect.
à System.Number.StringToNumber(String str, NumberStyles options,
     NumberBuffer
& number, NumberFormatInfo info, Boolean parseDecimal)
à System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
à System.Convert.ToInt32(String value)
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    14
```

Il s'agit tout simplement d'une erreur. Nous aurons l'occasion d'étudier plus en détail les exceptions dans les chapitres ultérieurs. Pour l'instant, on a juste besoin de voir que ceci ne nous convient pas. L'erreur est explicite : « Le format de la chaîne d'entrée est incorrect », mais cela se passe au moment de l'exécution et notre programme plante lamentablement. Nous verrons dans le chapitre sur les exceptions comment gérer les erreurs.

En interne, la méthode `Convert.ToInt32()` utilise une autre méthode pour faire la conversion, il s'agit de la méthode `int.Parse()`. Donc plutôt que d'utiliser une méthode qui en appelle une autre, nous pouvons nous en servir directement, par exemple :

```
string chaineAge = "20";
int age = int.Parse(chaineAge); // age vaut 20
```

Bien sûr, il se passera exactement la même chose que précédemment quand la chaîne ne représentera pas un entier correct.

Il existe par contre une autre façon de convertir une chaîne en entier qui ne provoque pas d'erreur quand le format n'est pas correct et qui nous informe si la conversion s'est bien passée ou non, c'est la méthode `int.TryParse()` qui s'utilise ainsi :

```
string chaineAge = "ab20cd";
int age;
if (int.TryParse(chaineAge, out age))
{
    Console.WriteLine("La conversion est possible, age vaut " +
        age);
}
else
{
    Console.WriteLine("Conversion impossible");
}
```

Nous aurons alors :

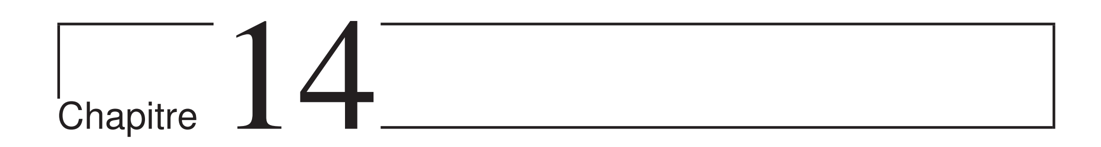
```
Conversion impossible
```

La méthode `int.TryParse` nous renvoie vrai si la conversion est bonne, et faux sinon. Nous pouvons donc effectuer une action en fonction du résultat grâce aux `if/else`. On passe en paramètres la chaîne à convertir et la variable où l'on veut stocker le résultat de la conversion. Le mot-clé `out` signifie juste que la variable est modifiée par la méthode.

Les méthodes que nous venons de voir `Convert.ToString()` ou `int.TryParse()` se déclinent en général pour tous les types de base, par exemple `double.TryParse()` ou `Convert.ToDecimal()`, etc.

En résumé

– Il est possible, avec le casting, de convertir la valeur d'un type dans un autre lorsqu'ils sont compatibles entre eux.
– Le casting explicite s'utilise en préfixant une variable par un type précisé entre parenthèses.
– Le framework .NET possède des méthodes permettant de convertir des types incompatibles entre eux s'ils sont sémantiquement proches.



Chapitre 14

Lire le clavier dans la console

Difficulté : 

Nous avons beaucoup écrit avec `Console.WriteLine()`, maintenant il est temps de savoir lire. En l'occurrence, il faut être capable de récupérer ce que l'utilisateur a saisi au clavier. En effet, un programme qui n'a pas d'interaction avec l'utilisateur, ce n'est pas vraiment intéressant. Vous imaginez un jeu où on ne fait que regarder ? Ça s'appelle une vidéo, c'est intéressant aussi, mais ce n'est pas pareil !

Aujourd'hui, il existe beaucoup d'interfaces de communication que l'on peut utiliser sur un ordinateur (clavier, souris, doigts, manettes,...). Dans ce chapitre nous allons regarder comment savoir ce que l'utilisateur a saisi au clavier dans une application console.



Lire une phrase

Lorsque nous lui en donnerons la possibilité, l'utilisateur de notre programme pourra saisir des choses grâce à son clavier. Nous pouvons lui demander son âge, s'il veut quitter l'application, lui permettre de saisir un mail, etc.

Il nous faut donc trouver un moyen lui permettant de saisir des caractères en tapant sur son clavier. Nous pourrons faire cela grâce à la méthode `Console.ReadLine` :

```
string saisie = Console.ReadLine();
```

Lorsque notre application rencontre cette instruction, elle se met en pause et attend une saisie de la part de l'utilisateur. La saisie s'arrête lorsque l'utilisateur valide ce qu'il a écrit avec la touche `Entrée`. Ainsi les instructions suivantes :

```
Console.WriteLine("Veuillez saisir une phrase et valider avec
    la touche \"Entrée\"");
string saisie = Console.ReadLine();
Console.WriteLine("Vous avez saisi : " + saisie);
```

produiront le message suivant :

```
Veuillez saisir une phrase et valider avec la touche "Entrée"
Bonjour à tous
Vous avez saisi : Bonjour à tous
```

Vous aurez remarqué que nous stockons le résultat de la saisie dans une variable de type chaîne de caractères. C'est bien souvent le seul type que nous aurons à notre disposition lors des saisies utilisateurs. Cela veut bien sûr dire que si vous avez besoin de manipuler l'âge de la personne, il faudra la convertir en entier. Par exemple :

```
bool ageIsValid = false;
int age = -1;
while (!ageIsValid)
{
    Console.WriteLine("Veuillez saisir votre âge");
    string saisie = Console.ReadLine();
    if (int.TryParse(saisie, out age))
        ageIsValid = true;
    else
    {
        ageIsValid = false;
        Console.WriteLine("L'âge que vous avez saisi est
            incorrect ...");
    }
}
Console.WriteLine("Votre âge est de : " + age);
```

Le programme insistera alors pour que l'utilisateur entre un nombre :

```
Veuillez saisir votre âge
abc
L'âge que vous avez saisi est incorrect...
Veuillez saisir votre âge
25
Votre âge est de : 25
```

Ce code est facile à comprendre maintenant que vous maîtrisez les boucles et les conditions. Nous commençons par initialiser nos variables. Ensuite, nous demandons à l'utilisateur de saisir son âge. Nous tentons de le convertir en entier. Si cela fonctionne, on pourra passer à la suite, sinon, tant que l'âge n'est pas valide, nous recommençons la boucle.



Il est primordial de toujours vérifier ce que saisit l'utilisateur, cela vous évitera beaucoup d'erreurs et de plantages intempestifs. On dit souvent qu'en informatique, il ne faut jamais faire confiance à l'utilisateur !

Lire un caractère

Il peut arriver que nous ayons besoin de ne saisir qu'un seul caractère. Pour cela, nous allons pouvoir utiliser la méthode `Console.ReadKey()`. Nous pourrons nous en servir par exemple pour faire une pause dans notre application. Le code suivant réclame l'attention de l'utilisateur avant de démarrer un calcul hautement important :

```
Console.WriteLine("Veuillez appuyer sur une touche pour démarrer le calcul ...");
Console.ReadKey(true);
int somme = 0;
for (int i = 0; i < 100; i++)
{
    somme += i;
}
Console.WriteLine(somme);
```

Notez que nous avons passé `true` en paramètre de la méthode afin d'indiquer au C# que nous ne souhaitions pas que notre saisie apparaisse à l'écran. Si le paramètre avait été `false` et que j'avais par exemple appuyé sur `H` pour démarrer le calcul, le résultat se serait vu précédé d'un disgracieux « H »...



D'ailleurs, comment faire pour savoir quelle touche a été saisie ?

Il suffit d'observer le contenu de la variable renvoyée par la méthode `Console.ReadKey`. Elle renvoie en l'occurrence une variable du type `ConsoleKeyInfo`. Nous pourrons

tester la valeur `Key` de cette variable qui est une énumération du type `ConsoleKey`. Par exemple :

```
Console.WriteLine("Voulez-vous continuer (O/N) ?");
ConsoleKeyInfo saisie = Console.ReadKey(true);
if (saisie.Key == ConsoleKey.O)
{
    Console.WriteLine("On continue ...");
}
else
{
    Console.WriteLine("On s'arrête ...");
}
```

Nous remarquons grâce à la complétion automatique que l'énumération `ConsoleKey` possède plein de valeurs, comme illustré à la figure 14.1.

Figure 14.1 – La structure `ConsoleKey` contient toutes les touches du clavier

Nous comparons donc à la valeur `ConsoleKey.O` qui représente la touche O.

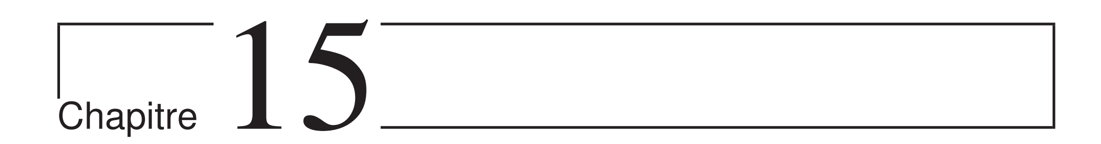
Notez que le type `ConsoleKeyInfo` est ce qu'on appelle une structure et que `Key` est une propriété de la structure. Nous reviendrons dans la partie suivante sur ce que cela veut vraiment dire. Pour l'instant, considérez juste qu'il s'agit d'une variable évoluée qui permet de contenir plusieurs choses…

Quand même, je trouve que c'est super de pouvoir lire les saisies de l'utilisateur ! C'est vital pour toute application qui se respecte. Aujourd'hui, dans une application moderne, il est primordial de savoir lire la position de la souris, réagir à des clics sur des boutons, ouvrir des fenêtres, etc. C'est quelque chose qu'il n'est pas possible à faire avec les programmes en mode console. N'oubliez pas que ces applications sont parfaites pour apprendre le C#, cependant il est inimaginable de réaliser une application digne de ce nom avec une application console.

Pour ce faire, on utilisera des systèmes graphiques adaptés, qui sortent du cadre d'étude de ce livre.

En résumé

- La méthode `Console.ReadLine` nous permet de lire des informations saisies par l'utilisateur au clavier.
- Il est possible de lire toute une phrase terminée par la touche `Entrée` ou seulement un unique caractère.



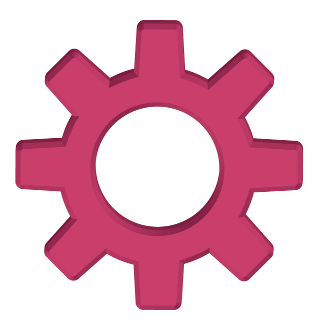
Chapitre 15

Utiliser le débogueur

Difficulté : 

Nous allons maintenant faire une petite pause. Le C# c'est bien, mais notre environnement de développement, Visual C# Express, peut faire beaucoup plus que sa fonction basique d'éditeur de fichiers. Il possède un outil formidable qui va nous permettre d'être très efficaces dans le débogage de nos applications et dans la compréhension de leur fonctionnement.

Il s'agit du débogueur. Découvrons vite à quoi il sert et comment il fonctionne !



À quoi ça sert ?

Fidèle à son habitude de nous simplifier la vie, Visual C# Express possède un débogueur. C'est un outil très pratique qui va permettre d'obtenir plein d'informations sur le déroulement d'un programme.

Il va permettre d'exécuter les instructions les unes après les autres, de pouvoir observer le contenu des variables, de revenir en arrière, bref, de savoir exactement ce qui se passe et surtout pourquoi tel bout de code ne fonctionne pas.

Pour l'utiliser, il faut que Visual C# Express soit en mode « débogage ». Je n'en ai pas encore parlé. Pour ce faire, il faut exécuter l'application en appuyant sur F5, ou alors passer par le menu `Déboguer` > `Démarrer le débogage`, ou encore cliquer sur le petit triangle vert dont l'icône rappelle un bouton de lecture (voir figure 15.1).

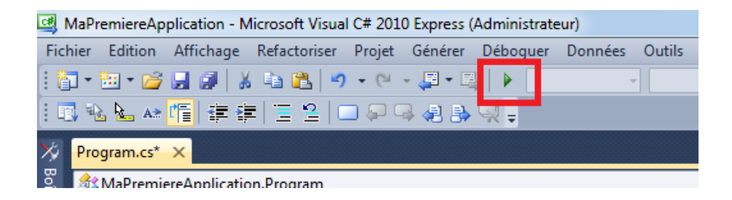


FIGURE 15.1 – Démarrer le débogage de l'application en cliquant sur le triangle vert

Pour étudier le débogueur, reprenons la dernière méthode du précédent TP :

```
static void Main(string[] args)
{
    Console.WriteLine(CalculSommeIntersection());
}

static int CalculSommeIntersection()
{
    List<int> multiplesDe3 = new List<int>();
    List<int> multiplesDe5 = new List<int>();

    for (int i = 1; i <= 100; i++)
    {
        if (i % 3 == 0)
            multiplesDe3.Add(i);
        if (i % 5 == 0)
            multiplesDe5.Add(i);
    }

    int somme = 0;
    foreach (int m3 in multiplesDe3)
    {
        foreach (int m5 in multiplesDe5)
        {
            if (m3 == m5)
```

```
                    somme += m3;
                }
            }
            return somme;
        }
```

Mettre un point d'arrêt et avancer pas à pas

Pour que le programme s'arrête sur un endroit précis et qu'il nous permette de voir ce qui se passe, il va falloir mettre des **points d'arrêt** dans notre code.

Pour mettre un point d'arrêt, il faut se positionner sur la ligne où nous souhaitons nous arrêter, par exemple la première ligne où nous appelons `Console.WriteLine` et appuyer sur F9. Nous pouvons également cliquer dans la marge à gauche pour produire le même résultat. Un point rouge s'affiche et indique qu'il y a un point d'arrêt à cet endroit, ainsi que vous pouvez le voir à la figure 15.2.

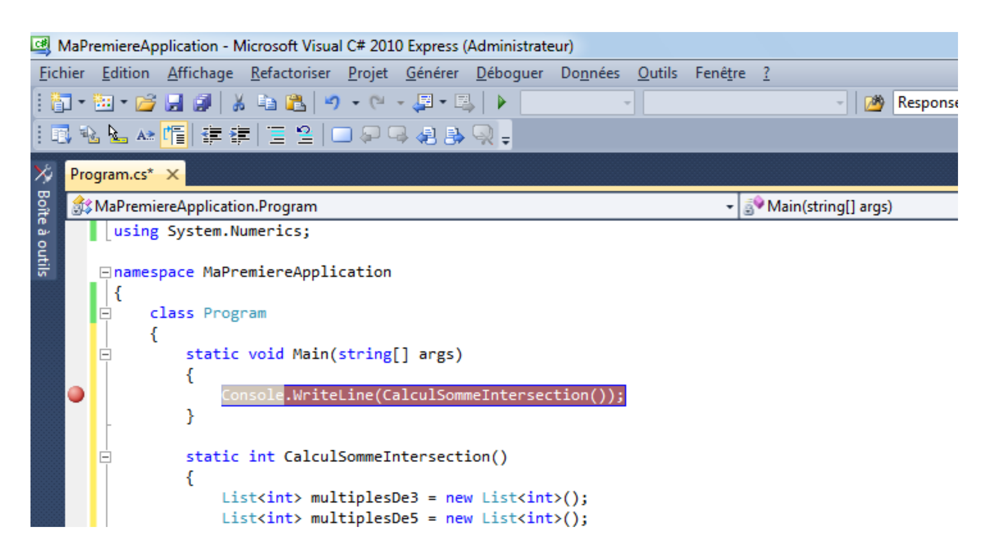


FIGURE 15.2 – Pose d'un point d'arrêt

Il est possible de mettre autant de points d'arrêt que nous le souhaitons, à n'importe quel endroit de notre code.

Lançons l'application avec F5. Nous pouvons voir que Visual C# Express s'arrête sur la ligne prévue en la surlignant en jaune (voir figure 15.3).

Nous en profitons pour remarquer au niveau du carré rouge, que le débogueur est en pause et qu'il est possible, soit de continuer l'exécution (triangle), soit de l'arrêter (carré), soit enfin de redémarrer le programme depuis le début (carré avec une flèche blanche dedans). Nous pouvons désormais exécuter notre code pas à pas, en nous servant des icônes à côté ou des raccourcis clavier indiqués à la figure 15.4.

Figure 15.3 – Visual C# Express est arrêté sur une ligne du code

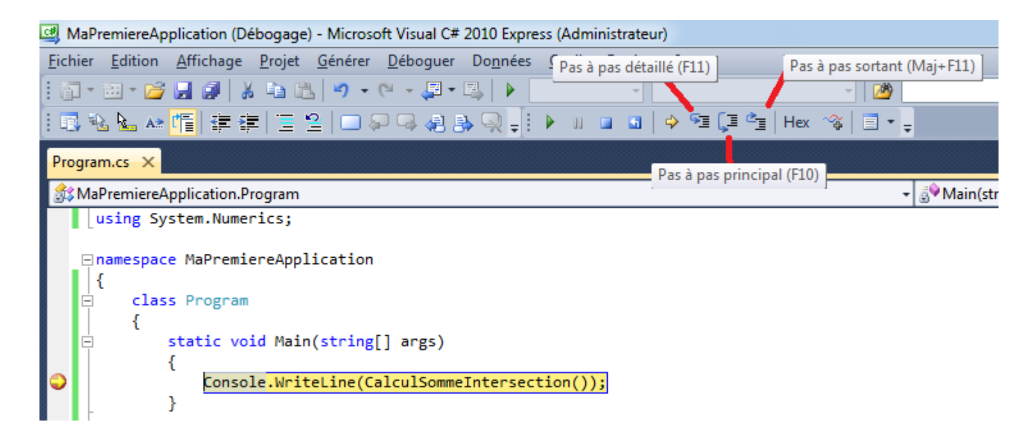


Figure 15.4 – Les boutons permettant d'avancer pas à pas dans l'application

Utilisons la touche `F10` pour continuer l'exécution du code pas à pas. La ligne suivante se trouve surlignée à son tour. Appuyons à nouveau sur la touche `F10` pour aller à l'instruction suivante ; il n'y en a plus : le programme se termine.

Vous me direz : « c'est bien beau, mais nous ne sommes pas passés dans la méthode `CalculSommeIntersection()` ». Eh oui, c'est parce que nous avons utilisé la touche `F10` qui est le pas à pas principal. Pour rentrer dans la méthode, il aurait fallu utiliser la touche `F11` qui est le pas à pas détaillé.

Si nous souhaitons que le programme se poursuive pour aller jusqu'au prochain point d'arrêt, il suffit d'appuyer sur le triangle (`play`) ou sur `F5`.

Relançons notre programme en mode débogage, et cette fois-ci, lorsque le débogueur s'arrête sur notre point d'arrêt, appuyons sur `F11` pour rentrer dans le corps de la méthode. Voilà, nous sommes dans la méthode `CalculSommeIntersection()`. Continuons à appuyer plusieurs fois sur `F10` afin de rentrer dans le corps de la boucle `for`.

Observer des variables

À ce moment-là du débogage, si nous passons la souris sur la variable `i`, qui est l'indice de la boucle, Visual C# Express va nous afficher une mini-information (voir figure 15.5).

Figure 15.5 – Visualisation du contenu d'une variable

Il nous indique que `i` vaut 1, ce qui est normal, car nous sommes dans la première itération de la boucle. Si nous continuons le parcours en appuyant sur `F10` plusieurs fois, nous voyons que la valeur de `i` augmente, conformément à ce qui est attendu. Maintenant, mettons un point d'arrêt (avec la touche `F9`) sur la ligne :

```
multiplesDe3.Add(i);
```

et poursuivons l'exécution en appuyant sur `F5`. Il s'arrête au moment où `i` vaut 3. Appuyons sur `F10` pour exécuter l'ajout de `i` à la liste et passons la souris sur la liste (voir figure 15.6).

Visual C# Express nous indique que la liste `multiplesDe3` a son `Count` qui vaut 1. Si nous cliquons sur le + pour déplier la liste, nous pouvons voir que 3 a été ajouté dans le premier emplacement de la liste (indice 0). Si nous continuons l'exécution plusieurs fois, nous voyons que les listes se remplissent au fur et à mesure (voir figure 15.7).

```
if (i % 3 == 0)
    multiplesDe3.Add(i);
if (i % 5 ==  ⊞ multiplesDe3 | Count = 1
    multiplesDe5.Add(i);
```

FIGURE 15.6 – Visualisation d'une liste

```
for (int i = 1; i <= 100; i++)
{
    if (i % 3 == 0)
        multiplesDe3.Add(i);
    if (i % 5 =  ⊟ multiplesDe3 | Count = 1
        multiplesI     [0]            | 3
                   ⊞ Affichage brut
}
```

FIGURE 15.7 – Visualisation d'un élément de la liste

Enlevez le point d'arrêt sur la ligne en appuyant à nouveau sur F9 et mettez un nouveau point d'arrêt sur la ligne :

```
int somme = 0;
```

Poursuivez l'exécution avec F5, la boucle est terminée, nous pouvons voir à la figure 15.8 que les listes sont complètement remplies !

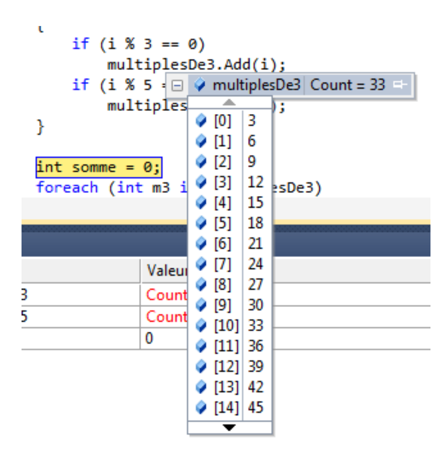


FIGURE 15.8 – Visualiser tous les éléments de la liste

Grâce au débogueur, nous pouvons voir vraiment tout ce qui nous intéresse, c'est une des grandes forces du débogueur et c'est un atout vraiment très utile pour comprendre ce qui se passe dans un programme (en général, ça se passe mal !).

Il est également possible de voir les variables locales en regardant en bas dans la fenêtre `Variables locales`. Dans cette fenêtre, vous pourrez observer les variables qui sont

couramment accessibles. Il existe également une fenêtre `Espion` qui permet, de la même façon, de surveiller une ou plusieurs variables précises (voir figure 15.9).

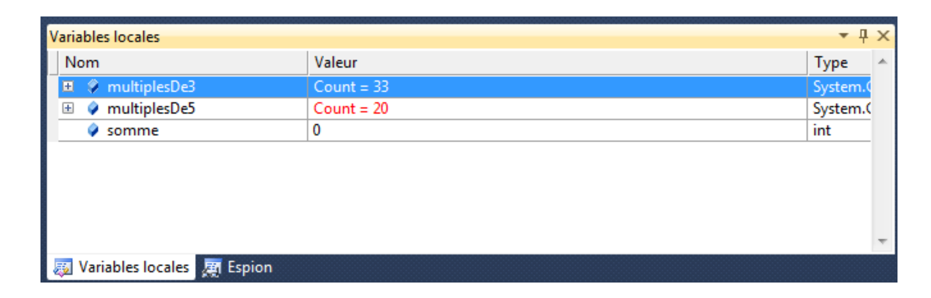


FIGURE 15.9 – La fenêtre des variables locales permet de voir le contenu des variables qui sont dans la portée

Revenir en arrière



Nom de Zeus, Marty ! On peut revenir dans le passé ?

C'est presque ça, Doc ! Si vous souhaitez réexécuter un bout de code, parce que vous n'avez pas bien vu ou que vous avez raté l'information qu'il vous fallait, vous pouvez forcer le débogueur à revenir en arrière dans le programme. Pour cela, vous avez deux solutions. Soit vous faites un clic droit à l'endroit souhaité et vous choisissez l'élément de menu **`Définir l'instruction suivante`**, comme indiqué à la figure 15.10 ; soit vous déplacez avec la souris la petite flèche jaune sur la gauche qui indique l'endroit où nous en sommes, comme l'illustre la figure 15.11.

Il faut faire attention, car ce retour en arrière n'en est pas complètement un. En effet, si vous revenez au début de la boucle qui calcule les multiples et que vous continuez l'exécution, vous verrez que la liste continue à se remplir. À la fin de la boucle, au lieu de contenir 33 éléments, la liste des multiples de 3 en contiendra 66. En effet, à aucun moment nous n'avons vidé la liste et donc le fait de réexécuter cette partie de code risque de provoquer des comportements inattendus. Ici, il vaudrait mieux revenir au début de la méthode afin que la liste soit de nouveau créée.

Même si ça ne vous saute pas aux yeux pour l'instant, vous verrez à l'usage que cette possibilité est bien pratique. D'autant plus quand vous aurez bien assimilé toutes les notions de portée de variable.



Il est important de noter que, bien que puissant, le débogueur de la version gratuite de Visual Studio est vraiment moins puissant que celui des versions payantes. Il y a un certain nombre de choses que l'on ne peut pas faire.

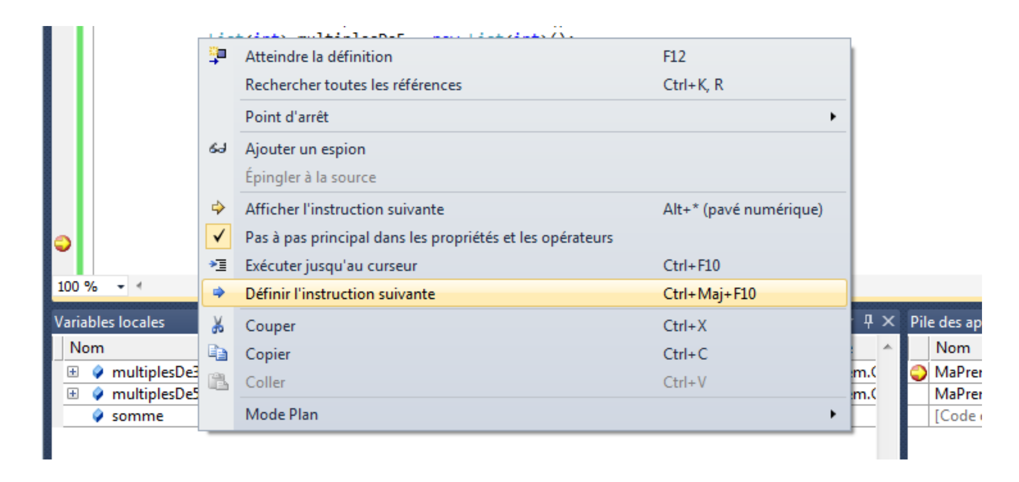


FIGURE 15.10 – Revenir en arrière dans l'application

```
        multiplesDe5.Add(i);
}

int somme = 0;
foreach (int m3 in multiplesDe3)
```

FIGURE 15.11 – La flèche jaune indique la future ligne de code à exécuter

Mais rassurez-vous, celui-ci est quand même déjà bien avancé et permet de faire beaucoup de choses.

La pile des appels

Une autre fenêtre importante à regarder est la pile des appels. Elle permet d'indiquer où nous nous trouvons et par où nous sommes passés pour arriver à cet endroit-là.

Par exemple, si vous appuyez sur F11 et que vous rentrez dans la méthode `CalculSommeIntersection()`, vous pourrez voir dans la pile des appels que vous êtes dans la méthode `CalculSommeIntersection()`, qui a été appelée depuis la méthode `Main()` (voir figure 15.12).

Si vous cliquez sur la ligne du `Main()`, Visual C# Express vous ramène automatiquement à l'endroit où a été fait l'appel. Cette ligne est alors surlignée en vert pour bien faire la différence avec le surlignage en jaune qui est vraiment l'endroit où se trouve le débogueur. C'est très pratique quand on a beaucoup de méthodes qui s'appellent les unes à la suite des autres (voir figure 15.13).



La pile des appels est également affichée lorsqu'on rencontre une erreur, elle permettra d'identifier plus facilement où est le problème. Vous l'avez vu dans le chapitre sur les conversions entre des types incompatibles.

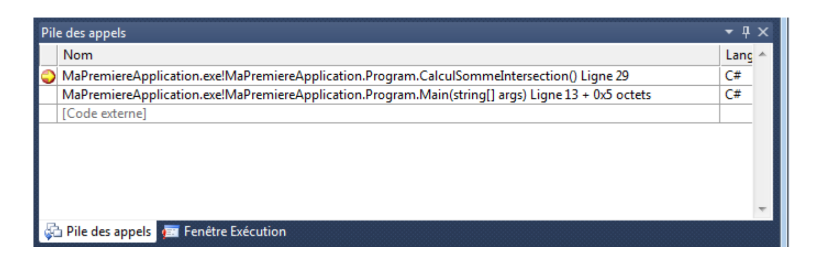


FIGURE 15.12 – La pile des appels

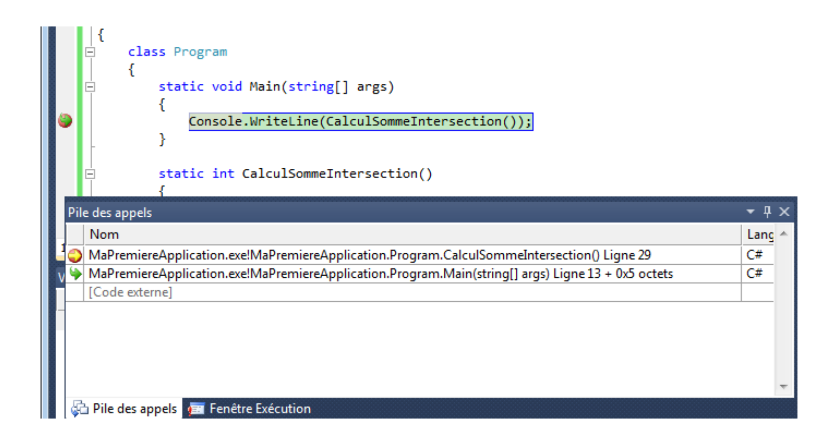


FIGURE 15.13 – Pointer la méthode qui encapsule la ligne courante

En tout cas, le débogueur est vraiment un outil à forte valeur ajoutée. Je ne vous ai présenté que le strict minimum nécessaire et indispensable. Mais croyez-moi, vous aurez l'occasion d'y revenir !

En résumé

- Le débogueur est un outil très puissant permettant d'inspecter le contenu des variables lors de l'exécution d'un programme.
- On peut s'arrêter à un endroit de notre application grâce à un point d'arrêt.
- Le débogueur permet d'exécuter son application pas à pas et de suivre son évolution.

Chapitre 16

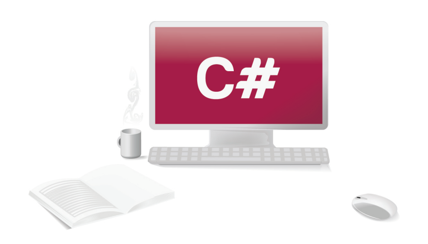
TP : le jeu du plus ou du moins

Difficulté : 

Waouh , nous augmentons régulièrement le nombre de choses que nous savons. C'est super. Nous commençons à être capables d'écrire des applications qui ont un peu plus de panache !

Enfin... moi, j'y arrive ! Et vous ? C'est ce que nous allons vérifier avec ce TP.

Savoir interagir avec son utilisateur est important. Voici donc un petit TP sous forme de création d'un jeu simple qui va vous permettre de vous entraîner. L'idée est de réaliser le jeu classique du plus ou du moins...



Instructions pour réaliser le TP

Je vous rappelle les règles. L'ordinateur calcule un nombre aléatoire et nous devons le deviner. À chaque saisie, il nous indique si le nombre saisi est plus grand ou plus petit que le nombre à trouver. Une fois trouvé, il nous indique en combien de coups nous avons réussi à trouver le nombre secret.

Pour ce TP, vous savez presque tout faire. Il ne vous manque que l'instruction pour obtenir un nombre aléatoire. La voici, cette instruction permet de renvoyer un nombre compris entre 0 et 100 (exclu). Ne vous attardez pas trop sur sa syntaxe, nous aurons l'occasion de comprendre exactement de quoi il s'agit dans la partie suivante :

```
int valeurATrouver = new Random().Next(0, 100);
```

Le principe est grosso modo le suivant : tant qu'on n'a pas trouvé la bonne valeur, nous devons en saisir une nouvelle. Dans ce cas, la console nous indique si la valeur est trop grande ou trop petite. Il faudra bien sûr incrémenter un compteur de coups à chaque essai.

N'oubliez pas de gérer le cas où l'utilisateur saisit n'importe quoi. Nous ne voudrions pas que notre premier jeu ait un bug qui fasse planter l'application !

Allez, je vous en ai trop dit. C'est à vous de jouer. Bon courage.

Correction

Voici ma correction de ce TP.

Bien sûr, il existe beaucoup de façons de réaliser ce petit jeu. S'il fonctionne, c'est que votre solution est bonne. Ma solution fonctionne, la voici :

```
static void Main(string[] args)
{
    int valeurATrouver = new Random().Next(0, 100);
    int nombreDeCoups = 0;
    bool trouve = false;
    Console.WriteLine("Veuillez saisir un nombre compris entre
        0 et 100 (exclu)");
    while (!trouve)
    {
        string saisie = Console.ReadLine();
        int valeurSaisie;
        if (int.TryParse(saisie, out valeurSaisie))
        {
            if (valeurSaisie == valeurATrouver)
                trouve = true;
            else
            {
                if (valeurSaisie < valeurATrouver)
                    Console.WriteLine("Trop petit ...");
```

```
                    else
                        Console.WriteLine("Trop grand ...");
                }
                nombreDeCoups++;
            }
            else
                Console.WriteLine("La valeur saisie est incorrecte,
                    veuillez recommencer ...");
        }
        Console.WriteLine("Vous avez trouvé en " + nombreDeCoups +
            " coup(s)");
}
```

On commence par obtenir un nombre aléatoire avec l'instruction que j'ai fournie dans l'énoncé. Nous avons ensuite les initialisations de variables. L'entier `nombreDeCoups` va permettre de stocker le nombre d'essai et le booléen `trouve` va permettre d'avoir une condition de sortie de boucle.

Notre boucle démarre et ne se terminera qu'une fois que le booléen `trouve` sera passé à vrai (`true`). Dans le corps de la boucle, nous demandons à l'utilisateur de saisir une valeur que nous essayons de convertir en entier. Si la conversion échoue, nous l'indiquons à l'utilisateur et nous recommençons notre boucle. Notez ici que je n'incrémente pas le nombre de coups, jugeant qu'il n'y a pas lieu de pénaliser le joueur parce qu'il a mal saisi ou qu'il a renversé quelque chose sur son clavier juste avant de valider la saisie.

Si en revanche, la conversion se passe bien, nous pouvons commencer à comparer la valeur saisie avec la valeur à trouver. Si la valeur est la bonne, nous passons le booléen à vrai, ce qui nous permettra de sortir de la boucle et de passer à la suite. Sinon, nous afficherons un message pour indiquer si la saisie est trop grande ou trop petite en fonction du résultat de la comparaison. Dans tous les cas, nous incrémenterons le nombre de coups. Enfin, en sortie de boucle, nous indiquerons sa victoire au joueur ainsi que le nombre de coups utilisés pour trouver le nombre secret :

```
Veuillez saisir un nombre compris entre 0 et 100 (exclu)
50
Trop petit..
75
Trop petit..
88
Trop grand..
81
Trop petit..
85
Trop grand..
83
Vous avez trouvé en 6 coup(s)
```

Aller plus loin

Il est bien sûr toujours possible d'améliorer le jeu. Nous pourrions par exemple ajouter un contrôle sur les bornes de la saisie. Ainsi, si l'utilisateur saisit un nombre supérieur ou égal à 100 ou inférieur à 0, nous pourrions lui rappeler les bornes du nombre aléatoire.

De même, à la fin, plutôt que d'afficher « coup(s) », nous pourrions tester la valeur du nombre de coups. S'il est égal à 1, on affiche « coup » au singulier, sinon « coups » au pluriel.

La boucle pourrait également être légèrement différente. Plutôt que de tester la condition de sortie sur un booléen, nous pourrions utiliser le mot-clé `break`. De même, nous pourrions alléger l'écriture avec le mot-clé `continue`. Par exemple :

```
static void Main(string[] args)
{
    int valeurATrouver = new Random().Next(0, 100);
    int nombreDeCoups = 0;
    Console.WriteLine("Veuillez saisir un nombre compris entre 0 et 100 (exclu)");
    while (true)
    {
        string saisie = Console.ReadLine();
        int valeurSaisie;
        if (!int.TryParse(saisie, out valeurSaisie))
        {
            Console.WriteLine("La valeur saisie est incorrecte, veuillez recommencer ...");
            continue;
        }
        if (valeurSaisie < 0 || valeurSaisie >= 100)
        {
            Console.WriteLine("Vous devez saisir un nombre entre 0 et 100 exclu ...");
            continue;
        }
        nombreDeCoups++;
        if (valeurSaisie == valeurATrouver)
            break;
        if (valeurSaisie < valeurATrouver)
            Console.WriteLine("Trop petit ...");
        else
            Console.WriteLine("Trop grand ...");
    }
    if (nombreDeCoups == 1)
        Console.WriteLine("Vous avez trouvé en " + nombreDeCoups + " coup");
    else
        Console.WriteLine("Vous avez trouvé en " + nombreDeCoups + " coups");
}
```

Tout ceci est une question de goût. Je préfère personnellement la version précédente n'aimant pas trop les `break` et les `continue`. Mais après tout, chacun fait comme il préfère, l'important est que nous nous amusions à écrire le programme et à y jouer !

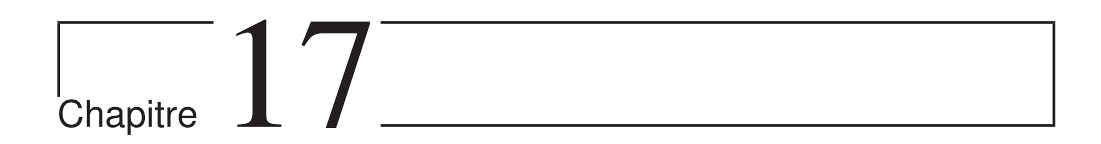
Voilà, ce TP est terminé. Vous avez pu voir finalement que nous étions tout à fait capables de réaliser de petites applications récréatives. Personnellement, j'ai commencé à m'amuser à faire de la programmation en réalisant toute sorte de petits programmes de ce genre.

Je vous encourage fortement à essayer de créer d'autres programmes par vous-mêmes. Plus vous vous entraînerez à faire des petits programmes simples et plus vous réussirez à appréhender les subtilités de ce que nous avons appris.

Bien sûr, plus tard, nous serons capables de réaliser des applications plus compliquées... Cela vous tente ? Alors continuons la lecture !

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 694204

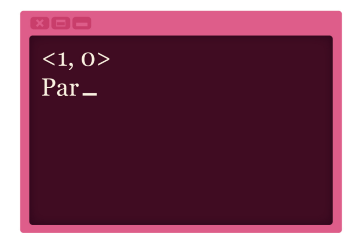


La ligne de commande

Difficulté : 

Sous ce nom un peu barbare se cache une fonctionnalité très présente dans nos usages quotidiens mais qui a tendance à être masquée à l'utilisateur lambda. Nous allons, dans un premier temps, voir ce qu'est exactement la ligne de commande et à quoi elle sert. Ensuite, nous verrons comment l'exploiter dans notre application avec le C#.

Notez que ce chapitre n'est pas essentiel mais qu'il pourra sûrement vous servir plus tard dans la création de vos applications.



Qu'est-ce que la ligne de commande ?

La ligne de commande, c'est ce qui nous permet d'exécuter nos programmes. Très présente à l'époque où Windows existait peu, elle a tendance à disparaître de nos utilisations. Mais pas le fond de son fonctionnement.

En général, la ligne de commande sert à passer des arguments à un programme. Par exemple, pour ouvrir un fichier texte avec le bloc-notes (`notepad.exe`), on peut le faire de deux façons différentes. Soit on ouvre le bloc-notes et on va dans le menu `Fichier` > `Ouvrir` puis on va chercher le fichier pour l'ouvrir. Soit on utilise la ligne de commande pour l'ouvrir directement.

Pour ce faire, il suffit d'utiliser la commande : `notepad c:\test.txt` et le fichier `c:\test.txt` s'ouvre directement dans le bloc-notes.

Ce qu'il s'est passé derrière, c'est que nous avons demandé d'exécuter le programme `notepad.exe` avec le paramètre `c:\test.txt`. Le programme notepad a analysé sa ligne de commande, il y a trouvé un paramètre et il a ouvert le fichier correspondant.

Superbe ! Nous allons apprendre à faire pareil !

Passer des paramètres en ligne de commande

La première chose est de savoir comment faire pour passer des paramètres en ligne de commande. Plusieurs solutions existent. On peut, par exemple, exécuter la commande que j'ai écrite plus haut depuis le menu `Démarrer` > `Exécuter` ou la combinaison touche Windows + `R` (voir la figure 17.1).

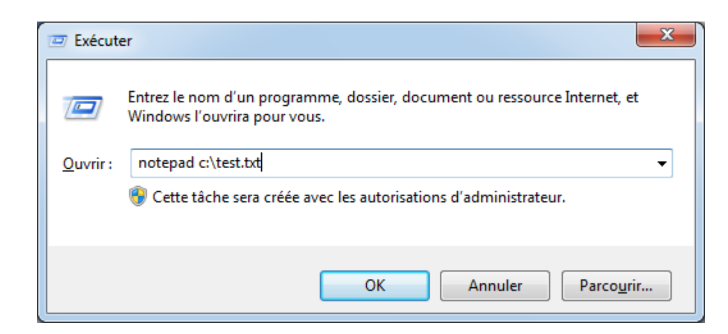


FIGURE 17.1 – Exécuter une application avec la ligne de commande

On peut également le faire depuis une invite de commande, en allant dans : `Menu Démarrer` > `Accessoires` > `Invite de commande`.

Ceci nous ouvre une console noire, comme celle que l'on connaît bien et dans laquelle nous pouvons taper des instructions (voir figure 17.2).

Mais nous, ce qui nous intéresse surtout, c'est de pouvoir le faire depuis Visual C# Express afin de pouvoir passer des arguments à notre programme. Pour ce faire, on va

```
Administrateur : Invite de commandes
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\Nico>notepad c:\test.txt
```

FIGURE 17.2 – Exécuter une application depuis l'invite de commande

aller dans les propriétés de notre projet (clic droit sur le projet, `Propriétés`) puis nous allons dans l'onglet `Déboguer` et nous voyons une zone de texte permettant de mettre des arguments à la ligne de commande. Rajoutons par exemple « Bonjour Nico » (voir figure 17.3).

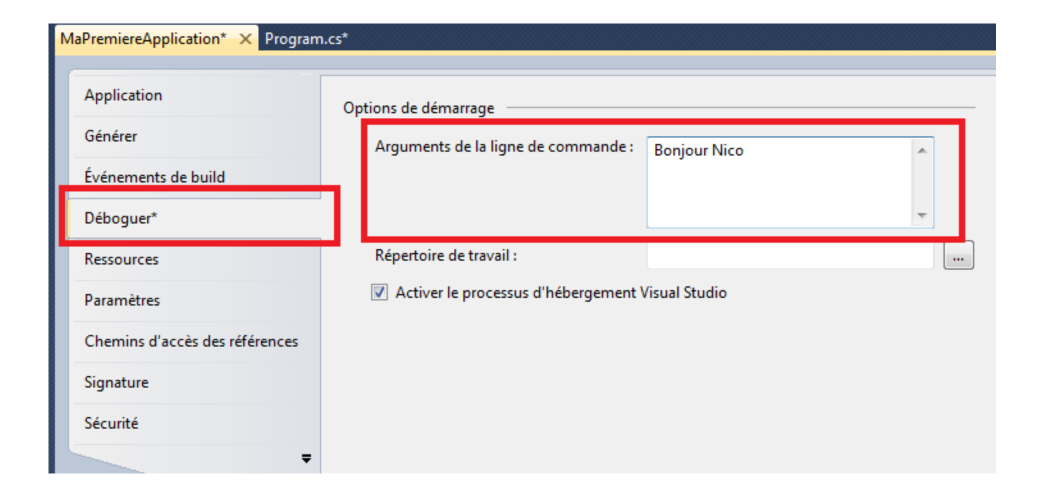


FIGURE 17.3 – Modifier les arguments de la ligne de commande

Voilà, maintenant lorsque nous exécuterons notre application, Visual C# Express lui passera les arguments que nous avons définis en paramètre de la ligne de commande.

À noter que dans ce cas, les arguments `Bonjour Nico` ne seront valables que lorsque nous exécuterons l'application à travers Visual C# Express. Évidemment, si nous exécutons notre application par l'invite de commande, nous aurons besoin de repasser les arguments au programme pour qu'il puisse les exploiter.

C'est bien ! Sauf que pour l'instant, ça ne nous change pas la vie ! Il faut apprendre à traiter ces paramètres. . .

Lire la ligne de commande

On peut lire le contenu de la ligne de commande de deux façons différentes. Vous vous rappelez de la méthode `Main()` ? On a vu que Visual C# Express générait cette méthode avec des paramètres. Eh bien ces paramètres, vous ne devinerez jamais ! Ce sont les paramètres de la ligne de commande. Oh joie !

```
static void Main(string[] args)
{
}
```

La variable `args` est un tableau de chaînes de caractères. Sachant que chaque paramètre est délimité par des espaces, nous retrouverons chacun des paramètres à un indice du tableau différent.

Ce qui fait que si j'utilise le code suivant :

```
foreach (string parametre in args)
{
    Console.WriteLine(parametre);
}
```

et que je lance mon application avec les paramètres définis précédemment, je vais obtenir :

```
Bonjour
Nico
```

Et voilà, nous avons récupéré les paramètres de la ligne de commande, il ne nous restera plus qu'à les traiter dans notre programme. Comme prévu, nous obtenons deux paramètres. Le premier est la chaîne de caractères « Bonjour », le deuxième est la chaîne de caractères « Nico ». N'oubliez pas que c'est le caractère d'espacement qui sert de délimiteur entre les paramètres.

L'autre façon de récupérer la ligne de commande est d'utiliser la méthode `Environment.GetCommandLineArgs()`. Elle renvoie un tableau contenant les paramètres, comme ce qui est passé en paramètres à la méthode `Main`. La seule différence, c'est que dans le premier élément du tableau, nous trouverons le chemin complet de notre programme. Ceci peut être utile dans certains cas. Si cela n'a aucun intérêt pour vous, il suffira de commencer la lecture à partir de l'indice numéro 1.

L'exemple suivant, affiche toutes les valeurs du tableau :

```
foreach (string parametre in Environment.GetCommandLineArgs())
{
    Console.WriteLine(parametre);
}
```

Ce qui donne :

```
C:\Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\bin\Release\
    MaPremiereApplication.exe
Bonjour
Nico
```



Attention : si vous devez accéder à un indice précis du tableau, vérifiez bien que la taille du tableau le permet. N'oubliez pas que si le tableau contient un seul élément et que vous essayez d'accéder au deuxième élément, alors il y aura une erreur.



Tu as dit que c'était le caractère d'espacement qui permettait de délimiter les paramètres. J'ai essayé de rajouter le paramètre `C:\Program Files\test.txt` à ma ligne de commande afin de changer l'emplacement du fichier, mais je me retrouve avec deux paramètres au lieu d'un seul, c'est normal ?

Eh oui, il y a un espace entre `Program` et `Files`. L'astuce est de passer le paramètre entre guillemets, de cette façon : `"C:\Program Files\test.txt"`. Nous aurons un seul paramètre dans la ligne de commande à la place de deux. Comme par exemple à la figure 17.4.

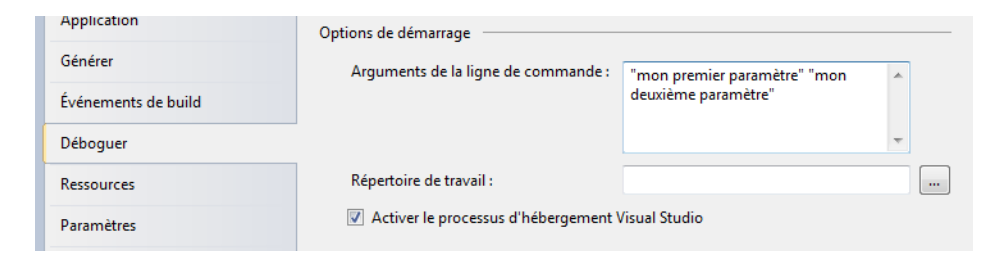
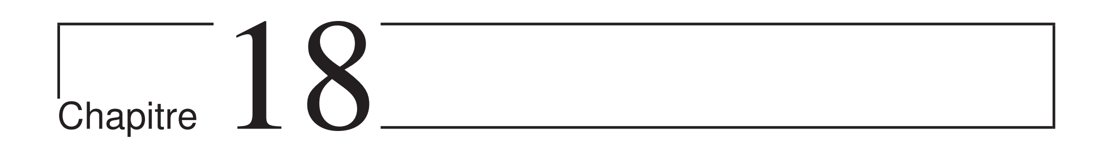


Figure 17.4 – Modifier les arguments de la ligne de commande

En résumé

- On peut passer des paramètres à une application en utilisant la ligne de commande.
- Le programme C# peut lire et interpréter ces paramètres.
- Ces paramètres sont séparés par des espaces.



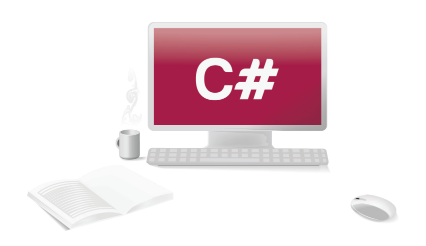
TP : calculs en ligne de commande

Difficulté : 

Et voici un nouveau TP qui va nous permettre de récapituler un peu tout ce que nous avons vu. Au programme, des `if`, des `switch`, des méthodes, des boucles et... de la ligne de commande bien sûr.

Grâce à nos connaissances grandissantes, nous arrivons à augmenter la difficulté de nos exercices et c'est une bonne chose. Vous verrez que dans le moindre petit programme, vous aurez besoin de toutes les notions que nous avons apprises.

Allez, ne traînons pas trop et à vous de travailler !



Instructions pour réaliser le TP

L'objectif de ce TP est de réaliser un petit programme qui permet d'effectuer une opération mathématique à partir de la ligne de commande.

Il va donc falloir écrire un programme que l'on pourra appeler avec des paramètres. En imaginant que le programme s'appelle `MonProgramme`, on pourrait l'utiliser ainsi pour faire la somme de deux nombres :

– `MonProgramme` somme 2 5

Ce qui renverra bien sûr 7.

Pour effectuer une multiplication, nous pourrons faire :

– `MonProgramme` multiplication 2 5

et nous obtiendrons 10.

Enfin, pour calculer une moyenne, nous pourrons faire :

– `MonProgramme` moyenne 1 2 3 4 5

ce qui renverra 3.

Les règles sont les suivantes :

– il doit y avoir deux et seulement deux nombres composant l'addition et la multiplication ;
– la moyenne peut contenir autant de nombres que souhaité ;
– si le nombre de paramètres est incorrect, alors le programme affichera un message d'aide explicite ;
– si le nombre attendu n'est pas un double correct, on affichera le message d'aide ;

Plutôt simple, non ? Alors, à vous de jouer, il n'y a pas de subtilité.

Correction

STOP ! Je relève les copies.

J'ai dit qu'il n'y avait pas de subtilité, mais j'ai un peu menti en fait. Je l'avoue, c'était pour que vous alliez au bout du développement sans aide.

Pour vérifier que vous avez trouvé la subtilité, essayez votre programme avec les paramètres suivants :

```
MonProgramme addition 5,5 2,5
```



Obtenez-vous 8 ?

C'était à peu près la seule subtilité, il fallait juste savoir qu'un nombre à virgule s'écri-

vait avec une virgule plutôt qu'avec un point. Si nous écrivons 2.5, alors le C# ne sait pas faire la conversion.



En vrai, c'est un peu plus compliqué que ça et nous le verrons dans un prochain chapitre, mais l'écriture du nombre à virgule est dépendante de ce qu'on appelle la **culture courante** que l'on peut modifier dans les paramètres `Horloge, langue et région` du panneau de configuration de Windows. Si l'on change le « format de la date, de l'heure ou des nombres » on change la culture courante, et la façon dont les nombres à virgule (et autres) sont gérés est différente.

Quoi qu'il en soit, voici ma correction du TP :

```
static void Main(string[] args)
{
    if (args.Length == 0)
    {
        AfficherAide();
    }
    else
    {
        string operateur = args[0];
        switch (operateur)
        {
            case "addition":
                Addition(args);
                break;
            case "multiplication":
                Multiplication(args);
                break;
            case "moyenne":
                Moyenne(args);
                break;
            default:
                AfficherAide();
                break;
        }
    }
}

static void AfficherAide()
{
    Console.WriteLine("Utilisez l'application de la manière
        suivante :");
    Console.WriteLine("MonProgamme addition 2 5");
    Console.WriteLine("MonProgamme multiplication 2 5");
    Console.WriteLine("MonProgamme moyenne 2 5 10 11");
}

```

```
static void Addition(string[] args)
{
    if (args.Length != 3)
    {
        AfficherAide();
    }
    else
    {
        double somme = 0;
        for (int i = 1; i < args.Length; i++)
        {
            double valeur;
            if (!double.TryParse(args[i], out valeur))
            {
                AfficherAide();
                return;
            }
            somme += valeur;
        }
        Console.WriteLine("Résultat de l'addition : " + somme);
    }
}

static void Multiplication(string[] args)
{
    if (args.Length != 3)
    {
        AfficherAide();
    }
    else
    {
        double resultat = 1;
        for (int i = 1; i < args.Length; i++)
        {
            double valeur;
            if (!double.TryParse(args[i], out valeur))
            {
                AfficherAide();
                return;
            }
            resultat *= valeur;
        }
        Console.WriteLine("Résultat de la multiplication : " +
            resultat);
    }
}

static void Moyenne(string[] args)
{
    double total = 0;
```

```
    for (int i = 1; i < args.Length; i++)
    {
        double valeur;
        if (!double.TryParse(args[i], out valeur))
        {
            AfficherAide();
            return;
        }
        total += valeur;
    }
    total = total / (args.Length - 1);
    Console.WriteLine("Résultat de la moyenne : " + total);
}
```

Disons que c'est plus long que difficile…

Oui, je sais, lors de l'addition et de la multiplication, j'ai parcouru l'ensemble des paramètres alors que j'aurais pu utiliser uniquement `args[1]` et `args[2]`. L'avantage, c'est que si je supprime la condition sur le nombre de paramètres, alors mon addition pourra fonctionner avec autant de nombres que je le souhaite… Vous aurez remarqué que je commence mon parcours à l'indice 1, d'où la pertinence de l'utilisation de la boucle `for` plutôt que de la boucle `foreach`.

Notez quand même l'utilisation du mot-clé `return` afin de sortir prématurément de la méthode en cas de problème.

Évidemment, il y a plein de façons de réaliser ce TP, la mienne en est une, il y en a d'autres.

J'espère que vous aurez fait attention au calcul de la moyenne. Nous retirons 1 à la longueur du tableau pour obtenir le nombre de paramètres. Il faut donc faire attention à l'ordre des parenthèses afin que C# fasse d'abord la soustraction avant la division, alors que sans ça, la division est prioritaire sur la soustraction. Sinon, c'est l'erreur de calcul assurée, indigne de notre superbe application !

Aller plus loin

Ce code est très bien. Si, si, c'est moi qui l'ai fait !

Sauf qu'il y a quand même un petit problème… d'ordre architectural.

Il a été plutôt simple à écrire en suivant l'énoncé du TP mais on peut faire encore mieux. On peut le rendre plus facilement maintenable en l'organisant différemment.

Observez le code suivant :

```
static void Main(string[] args)
{
    bool ok = false;
    if (args.Length > 0)
    {
        string operateur = args[0];
```

```
        switch (operateur)
        {
            case "addition":
                ok = Addition(args);
                break;
            case "multiplication":
                ok = Multiplication(args);
                break;
            case "moyenne":
                ok = Moyenne(args);
                break;
        }
    }

    if (!ok)
        AfficherAide();
}

static void AfficherAide()
{
    Console.WriteLine("Utilisez l'application de la manière
        suivante :");
    Console.WriteLine("MonProgamme addition 2 5");
    Console.WriteLine("MonProgamme multiplication 2 5");
    Console.WriteLine("MonProgamme moyenne 2 5 10 11");
}

static bool Addition(string[] args)
{
    if (args.Length != 3)
        return false;

    double somme = 0;
    for (int i = 1; i < args.Length; i++)
    {
        double valeur;
        if (!double.TryParse(args[i], out valeur))
            return false;
        somme += valeur;
    }

    Console.WriteLine("Résultat de l'addition : " + somme);
    return true;
}

static bool Multiplication(string[] args)
{
    if (args.Length != 3)
        return false;
    double resultat = 1;
```

```
    for (int i = 1; i < args.Length; i++)
    {
        double valeur;
        if (!double.TryParse(args[i], out valeur))
            return false;
        resultat *= valeur;
    }
    Console.WriteLine("Résultat de la multiplication : " +
        resultat);
    return true;
}

static bool Moyenne(string[] args)
{
    double total = 0;
    for (int i = 1; i < args.Length; i++)
    {
        double valeur;
        if (!double.TryParse(args[i], out valeur))
            return false;
        total += valeur;
    }
    total = total / (args.Length - 1);
    Console.WriteLine("Résultat de la moyenne : " + total);
    return true;
}
```

Nous n'affichons l'aide qu'à un seul endroit en utilisant le fait que les méthodes renvoient un booléen indiquant si l'opération a été possible ou pas.

D'une manière générale, il est important d'essayer de rendre son code le plus maintenable possible, comme nous l'avons fait ici.

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 846482

Troisième partie

Le C#, un langage orienté objet

Chapitre 19

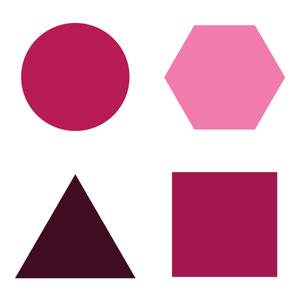
Introduction à la programmation orientée objet

Difficulté : 

Dans ce chapitre, nous allons essayer de décrire ce qu'est la programmation orientée objet (abrégée souvent en POO). Je dis bien « essayer », car pour être complètement traitée, la POO nécessiterait qu'on lui dédie un ouvrage entier !

Nous allons tâcher d'aller à l'essentiel et de rester proche de la pratique. Ce n'est pas très grave si tous les concepts abstraits ne sont pas parfaitement appréhendés : il est impossible d'apprendre la POO en deux heures. Cela nécessite une longue pratique, beaucoup d'empirisme et des approfondissements théoriques.

Ce qui est important ici, c'est de comprendre les notions de base et de pouvoir les utiliser dans de petits programmes. Après cette mise au point, attaquons sans plus tarder la programmation orientée objet !



Qu'est-ce qu'un objet ?

Vous avez pu voir précédemment que j'ai utilisé de temps en temps le mot « objet » et que le mot-clé `new` est apparu comme par magie... Il a été difficile de ne pas trop en parler et il est temps d'en savoir un peu plus.



Alors, qu'est-ce qu'un objet ?

Si on prend le monde réel (si, si, vous allez voir, vous connaissez...), nous sommes entourés d'objets : une chaise, une table, une voiture, etc. Ces objets forment un tout.

- Ils possèdent des propriétés (la chaise possède quatre pieds, elle est de couleur bleue, etc.).
- Ces objets peuvent faire des actions (la voiture peut rouler, klaxonner, etc.).
- Ils peuvent également interagir entre eux (l'objet conducteur démarre la voiture, l'objet voiture fait tourner l'objet volant, etc.).

Il faut bien faire attention à distinguer ce qu'est l'objet et ce qu'est la définition d'un objet.

La définition de l'objet (ou structure de l'objet) permet d'indiquer ce qui compose un objet, c'est-à-dire quelles sont ses propriétés, ses actions etc. Comme par exemple, le fait qu'une chaise ait des pieds ou qu'on puisse s'asseoir dessus. Par contre, l'objet chaise est bien concret. On peut donc avoir plusieurs objets chaise : on parle également d'instances. Les objets chaise, ce sont bien ceux, concrets, que l'on voit devant nous autour de l'objet table, pour démarrer une partie de belote. On peut faire l'analogie avec notre dictionnaire qui nous décrit ce qu'est une chaise. Le dictionnaire décrit en quoi consiste l'objet, et l'instance de l'objet représente le concret associé à cette définition. Chaque objet a sa propre vie et diffère d'un autre. Nous pouvons avoir une chaise bleue, une autre rouge, une autre avec des roulettes, une cassée...

Vous voyez, finalement, la notion d'objet est plutôt simple quand on la ramène à ce qu'on connaît déjà !

Sachant qu'un objet en programmation, c'est comme un objet du monde réel, mais ce n'est pas forcément restreint au matériel. Un chien est un objet. Des concepts comme l'amour ou une idée sont également des objets, tandis qu'on ne dirait pas cela dans le monde réel.

En conclusion :

- La définition (ou structure) d'un objet est un concept abstrait, comme une définition dans le dictionnaire. Cette définition décrit les caractéristiques d'un objet (la chaise a des pieds, l'homme a des jambes, etc.). Cette définition est unique, comme une définition dans le dictionnaire.
- Un objet ou une instance est la réalisation concrète de la structure de l'objet. On peut avoir de multiples instances, comme les cent voitures sur le parking devant chez moi. Elles peuvent avoir des caractéristiques différentes (une voiture bleue, une

voiture électrique, une voiture à cinq portes, etc.).

L'encapsulation

Le fait de concevoir une application comme un système d'objets interagissant entre eux apporte une certaine souplesse et une forte abstraction.

Prenons un exemple tout simple : la machine à café du bureau. Nous insérons nos pièces dans le monnayeur, choisissons la boisson et nous nous retrouvons avec un gobelet de la boisson commandée. Nous nous moquons complètement de savoir comment cela fonctionne à l'intérieur et nous pouvons complètement ignorer si le café est en poudre, en grain, comment l'eau est ajoutée, chauffée, comment le sucre est distribué, etc. Tout ce qui nous importe, c'est que le fait de mettre des sous dans la machine nous donne un café qui va nous permettre d'attaquer la journée.

Voilà un bel exemple de programmation orientée objet. Nous manipulons un objet `MachineACafe` qui a des propriétés (allumée/éteinte, présence de café, présence de gobelet,...) et qui sait faire des actions (`AccepterMonnaie`, `DonnerCafe`,...). Et c'est tout ce que nous avons besoin de savoir : on se moque du fonctionnement interne, peu importe ce qui se passe à l'intérieur, notre objet nous donne du café, point !

C'est ce qu'on appelle l'**encapsulation**. Cela permet de protéger l'information contenue dans notre objet et de le rendre manipulable uniquement par ses actions ou propriétés. Ainsi, l'utilisateur ne peut accéder ni au café ni au sucre et encore moins à la monnaie. Notre objet est ainsi protégé et fonctionne un peu comme une boîte noire.

L'intérêt est que si la personne qui entretient la machine met du café en grain à la place du café soluble, c'est invisible pour l'utilisateur qui n'a juste qu'à se soucier de mettre ses pièces dans la machine.

L'encapsulation protège donc les données de l'objet et son fonctionnement interne.

Héritage

Un autre élément important dans la programmation orientée objet, que nous allons aborder maintenant, est l'héritage.



Ah bon ? Les objets aussi peuvent mourir et transmettre leur patrimoine ?

Eh bien c'est presque comme en droit, à part que l'objet ne meurt pas et qu'il n'y a pas de taxe sur l'héritage. C'est-à-dire qu'un objet dit « père » peut transmettre certaines de ses caractéristiques à un autre objet dit « fils ».

Pour cela, on pourra définir une relation d'héritage entre eux.

S'il y a une relation d'héritage entre un objet père et un objet fils, alors **l'objet fils**

hérite de l'objet père. On dit également que l'objet fils est une **spécialisation** de l'objet père ou qu'**il dérive de l'objet père**.

En langage plus courant, on peut également dire que l'objet fils est « une sorte » d'objet père.



Je ne suis pas sûr de comprendre, peux-tu donner des exemples ?

Alors, prenons par exemple l'objet `chien` et imaginons ses caractéristiques tirées du monde réel en utilisant l'héritage :

- l'objet `chien` est une sorte d'objet `mammifere` ;
- l'objet `mammifere` est une sorte d'objet `animal` ;
- l'objet `animal` est une sorte d'objet `etreVivant`.

Chaque père est un peu plus général que son fils. Et inversement, chaque fils est un peu plus spécialisé que son père. Avec l'exemple du dessus, un mammifère est un peu plus général qu'un chien, l'être vivant étant encore plus général qu'un mammifère.

Il est possible pour un père d'avoir plusieurs fils, par contre, l'inverse est impossible, un fils ne peut pas avoir plusieurs pères. Eh oui, c'est triste mais c'est comme ça, c'est le règne du père célibataire avec plusieurs enfants à charge !

Ainsi, un objet `chat` peut également être un fils de l'objet `mammifere`. Un objet `vegetal` peut également être fils de l'objet `etreVivant`.

Ce qu'on peut reproduire sur le schéma suivant (voir figure 19.1) où chaque bulle représente un objet et chaque flèche représente l'héritage entre les objets.

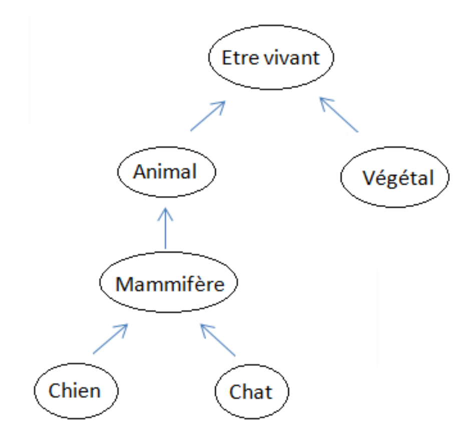


FIGURE 19.1 – L'héritage entre les objets

On peut définir une sorte de hiérarchie entre les objets, un peu comme on le ferait avec un arbre généalogique. La différence est qu'un objet héritant d'un autre peut obtenir certains ou tous les comportements de l'objet qu'il spécialise, alors qu'un enfant n'hérite

pas forcément des yeux bleus de sa mère ou du côté bougon de son grand-père, le hasard de la nature faisant le reste.

Pour bien comprendre cet héritage de comportement, empruntons à nouveau les exemples au monde réel.

- L'être vivant peut, par exemple, faire l'action « vivre ».
- Le mammifère possède des yeux.
- Le chien, qui est une sorte d'être vivant et une sorte de mammifère, peut également faire l'action « vivre » et aura des yeux.
- Le chat, qui est une autre sorte d'être vivant, peut lui aussi faire l'action « vivre » et aura également des yeux.

On voit bien ici que le chat et le chien héritent des comportements de leurs parents et grands-parents en étant capables de vivre et d'avoir des yeux.

Par contre, l'action « aboyer » est spécifique au chien. Ce qui veut dire que ni le chat ni le dauphin ne seront capables d'aboyer. Il n'y a que dans les dessins animés de Tex Avery que ceci est possible ! Évidemment, il n'y a pas de notion d'héritage entre le chien et le chat et l'action d'aboyer est définie au niveau du comportement du chien.

Ceci implique également que seul un objet qui est une sorte de chien, par exemple l'objet labrador ou l'objet chihuahua, pourra hériter du comportement « aboyer », car il y a une relation d'héritage entre eux.

Finalement, c'est plutôt logique !

Rappelons juste avant de terminer ce paragraphe qu'un objet ne peut pas hériter de plusieurs objets. Il ne peut hériter que d'un seul objet. Le C# ne permet pas ce qu'on appelle **l'héritage multiple**, a contrario d'autres langages comme le C++ par exemple.

Voilà globalement pour la notion d'héritage.

Je dis globalement, car il y a certaines subtilités que je n'ai pas abordées mais ce n'est pas trop grave ; vous verrez dans les chapitres suivants comment le C# utilise la notion d'héritage et ce qu'il y a vraiment besoin de savoir. Ne vous inquiétez pas si certaines notions sont encore un peu floues, vous comprendrez sûrement mieux grâce à la pratique.

Polymorphisme - Substitution

Le mot **polymorphisme** suggère qu'une chose peut prendre plusieurs formes. Sous ce terme un peu barbare se cachent plusieurs notions de l'orienté objet qui sont souvent source d'erreurs.

Je vais volontairement passer rapidement sur certains points qui ne vont pas nous servir pour me concentrer sur ceux qui sont importants pour ce livre.

En fait, on peut dire qu'une manifestation du polymorphisme est la capacité pour un objet de faire une même action avec différents types d'intervenants. C'est ce qu'on appelle le polymorphisme *ad hoc* ou le polymorphisme « paramétré ». Par exemple, notre objet voiture peut rouler sur la route, rouler sur l'autoroute, rouler sur la terre

si elle est équipée de pneus adéquats, rouler au fond de l'eau si elle est amphibie, etc.

Concrètement ici, je fais interagir un objet `voiture` avec un objet `autoroute` ou un objet `terre`... par l'action qui consiste à `rouler`. Cela peut paraître anodin décrit ainsi, mais nous verrons ce que cela implique avec le C#.

La **substitution** est une autre manifestation du polymorphisme. Il s'agit de la capacité d'un objet fils à redéfinir des caractéristiques ou des actions d'un objet père.

Prenons par exemple un objet mammifère qui sait faire l'action « se déplacer ». Les objets qui dérivent du mammifère peuvent potentiellement avoir à se déplacer d'une manière différente. Par exemple, l'objet homme va se déplacer sur ses deux jambes et donc différemment de l'objet dauphin qui se déplacera grâce à ses nageoires ou bien encore différemment de l'objet « homme accidenté » qui va avoir besoin de béquilles pour s'aider dans son déplacement.

Tous ces mammifères sont capables de se déplacer, mais chacun va le faire d'une manière différente. Ceci est donc possible grâce à la substitution qui permet de redéfinir un comportement hérité.

Ainsi, chaque fils sera libre de réécrire son propre comportement, si celui de son père ne lui convient pas.

Interfaces

Un autre concept important de la programmation orientée objet est la notion d'interface.



L'interface est un contrat que s'engage à respecter un objet. Il indique en général un comportement.

Prenons un exemple dans notre monde réel et connu : les prises de courant.

Elles fournissent de l'électricité à 220 V avec deux trous et (souvent) une prise de terre. Peu importe ce qu'il y a derrière, du courant alternatif de la centrale du coin, un transformateur, quelqu'un qui pédale... nous saurons à coup sûr que nous pouvons brancher nos appareils électriques car ces prises s'engagent à nous fournir du courant alternatif avec le branchement adéquat.

Elles respectent le contrat ; elles sont « branchables ».

Ce dernier terme est un peu barbare et peut faire mal aux oreilles. Mais vous verrez que suffixer des interfaces par un « able » est très courant et permet d'être plus précis sur la sémantique de l'interface. C'est également un suffixe qui fonctionne en anglais et les interfaces du framework .NET finissent pour la plupart par « able ».

À noter que les interfaces ne fournissent qu'un contrat, elles ne fournissent pas d'implémentation, c'est-à-dire pas de code C#. Ce n'est pas clair ? Les interfaces indiquent que les objets qui choisissent de respecter ce contrat auront forcément telle action ou telle caractéristique mais elles n'indiquent pas comment faire, c'est-à-dire qu'elles

n'ont pas de code C# associé. Chaque objet respectant cette interface (on parle d'objet **implémentant** une interface) sera responsable de coder la fonctionnalité associée au contrat.

Pour manipuler ces prises, nous pourrons utiliser cette interface en disant : « Allez hop, tous les branchables, venez par ici, on a besoin de votre courant ! » Peu importe que l'objet implémentant cette interface soit une prise murale ou une prise reliée à une dynamo, ou autre, nous pourrons manipuler ces objets par leur interface et donc brancher nos prises permettant d'alimenter nos appareils.

Contrairement à l'héritage, un objet est capable d'implémenter plusieurs interfaces. Par exemple, une pompe à chaleur peut être « chauffante » et « refroidissante » [1].

Nous en avons terminé avec la théorie sur les interfaces. Il est fort probable que vous ne saisissiez pas encore tout l'intérêt des interfaces ou ce qu'elles sont exactement. Nous allons y revenir avec des exemples concrets et vous verrez des utilisations d'interfaces dans le cadre du framework .NET qui vous éclaireront davantage.

À quoi sert la programmation orientée objet ?

Nous avons décrit plusieurs concepts de la programmation orientée objet mais nous n'avons pas encore dit à quoi elle allait nous servir.

En fait, on peut dire que la POO est une façon de développer une application qui consiste à représenter (on dit également « **modéliser** ») une application informatique sous la forme d'objets, ayant des propriétés et pouvant interagir entre eux.

La modélisation orientée objet est proche de la réalité ce qui fait qu'il sera relativement facile de modéliser une application de cette façon. De plus, les personnes non-techniques pourront comprendre et éventuellement participer à cette modélisation.

Cette façon de modéliser les choses permet également de découper une grosse application, généralement floue, en une multitude d'objets interagissant entre eux. Cela permet de découper un gros problème en plus petits afin de le résoudre plus facilement.

Utiliser une approche orientée objet améliore également la maintenabilité. Plus le temps passe et plus une application est difficile à maintenir. Il devient difficile de corriger des choses sans tout casser ou d'ajouter des fonctionnalités sans provoquer une régression par ailleurs. L'orienté objet nous aide ici à limiter la casse en proposant une approche où les modifications internes à un objet n'affectent pas tout le reste du code, grâce notamment à l'encapsulation.

Un autre avantage de la POO est la réutilisabilité. Des objets peuvent être réutilisés ou même étendus grâce à l'héritage. C'est le cas par exemple de la bibliothèque de classes du framework .NET que nous avons déjà utilisée. Cette bibliothèque nous fournit par exemple tous les objets permettant de construire des applications graphiques. Inutile de réinventer toute la mécanique pour gérer des fenêtres dans une application, le framework .NET sait déjà faire tout ça. Nous avons juste besoin d'utiliser un objet `fenêtre`,

1. Notez qu'en français, nous pourrons également utiliser le suffixe « ante ». En anglais, nous aurons plus souvent « able ».

dans lequel nous pourrons mettre un objet `bouton` et un objet `zone de texte`. Ces objets héritent tous des mêmes comportements, comme le fait d'être cliquable ou sélectionnable, etc.

De même, des composants tout faits et prêts à l'emploi peuvent être vendus par des entreprises tierces (systèmes de log, contrôles utilisateur améliorés, etc.).

Il faut savoir que la POO, c'est beaucoup plus que ça et nous en verrons des subtilités plus loin, mais comprendre ce qu'est un objet est globalement suffisant pour une grande partie de l'ouvrage.

En résumé

- L'approche orientée objet permet de modéliser son application sous la forme d'interactions entre objets.
- Les objets ont des propriétés et peuvent faire des actions.
- Ils masquent la complexité d'une implémentation grâce à l'encapsulation.
- Les objets peuvent hériter de fonctionnalités d'autres objets s'il y a une relation d'héritage entre eux.

Chapitre 20

Créer son premier objet

Difficulté : 

Ah , enfin un peu de concret et surtout de code. Dans ce chapitre, nous allons appliquer les notions que nous avons vues sur la programmation orientée objet pour continuer notre apprentissage du C#.

Même si vous n'avez pas encore appréhendé exactement où la POO pouvait nous mener, ce n'est pas grave. Les notions s'affineront au fur et à mesure de la lecture du livre. Il est temps pour nous de commencer à créer des objets, à les faire hériter entre eux, etc. Bref, à jouer, grâce à ces concepts, avec le C#.

Vous verrez qu'avec un peu de pratique tout s'éclaircira ; vous saisirez l'intérêt de la POO et comment être efficace avec le C#.



Tous les types C# sont des objets



Ça y est, il a fini avec son baratin qui donne mal à la tête ? Pourquoi tout ce blabla sur les objets ?

Parce que tout dans le C# est un objet. Comme déjà dit, une fenêtre Windows est un objet. Une chaîne de caractères est un objet. La liste que nous avons vue plus haut est un objet.

Nous avons vu que les objets possédaient des caractéristiques ; il s'agit de **propriétés**. Un objet peut également faire des actions ; ce sont des **méthodes**.

Suivant ce principe, une chaîne de caractères est un objet et possède des propriétés (par exemple sa longueur). De la même façon, il sera possible que les chaînes de caractères fassent des actions (par exemple se mettre en majuscules).

Nous allons voir plus bas qu'il est évidemment possible de créer nos propres objets (chat, chien, etc.) et que grâce à eux, nous allons enrichir les types qui sont à notre disposition. Un peu comme nous avons déjà fait avec les énumérations.

Voyons dès à présent comment faire, grâce aux classes.

Les classes

Dans le chapitre précédent, nous avons parlé des objets mais nous avons également parlé de la définition de l'objet, de sa structure. Eh bien, c'est exactement ce qu'est une classe.



Une classe est une manière de représenter un objet. Le C# nous permet de créer des classes.

Nous avons déjà pu voir une classe dans le code que nous avons utilisé précédemment et qui a été généré par Visual C# Express, la classe `Program`. Nous n'y avons pas fait trop attention, mais voilà à peu près à quoi elle ressemblait :

```
class Program
{
    static void Main(string[] args)
    {
    }
}
```

C'est elle qui contenait la méthode spéciale `Main()` qui sert de point d'entrée à l'application. Nous pouvons découvrir avec des yeux neufs le mot-clé `class` qui, comme son nom le suggère, permet de définir une classe, c'est-à-dire la structure d'un objet.

Rappelez-vous, les objets peuvent avoir des caractéristiques et faire des actions.

Ici, c'est exactement ce qui se passe. Nous avons défini la structure d'un objet `Program` qui contient une action : la méthode `Main()`. Vous aurez remarqué au passage que pour définir une classe, nous utilisons à nouveau les accolades permettant de créer un bloc de code qui délimite la classe.

Passons sur cette classe particulière et lançons-nous dans la création d'une classe qui nous servira à créer des objets. Par exemple, une classe `Voiture`. À noter qu'il est possible de créer une classe à plusieurs endroits dans le code, mais en général, nous utiliserons un nouveau fichier, du même nom que la classe, qui lui sera entièrement dédié. Une règle d'écriture commune à beaucoup de langages de programmation est que chaque fichier doit avoir une seule classe.

Faisons un clic droit sur notre projet pour ajouter une nouvelle classe, comme indiqué à la figure 20.1.

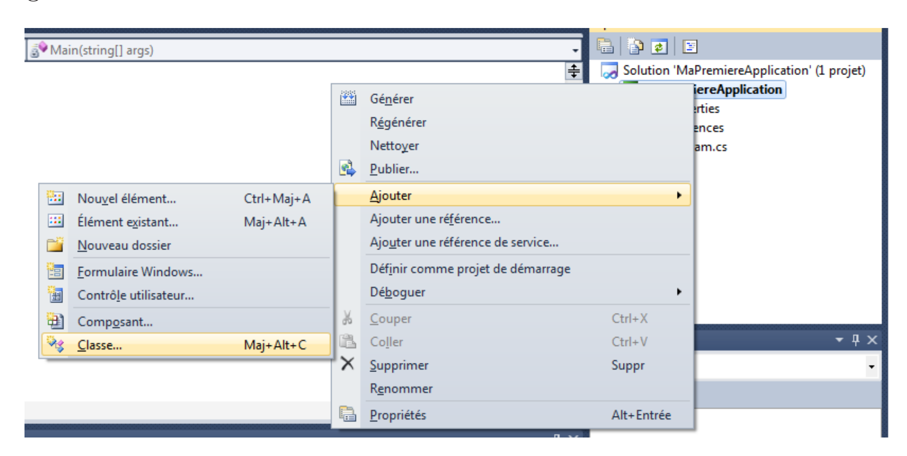


FIGURE 20.1 – Ajouter une classe

Visual C# Express nous ouvre sa fenêtre permettant de faire l'ajout d'un élément en se positionnant sur l'élément `Classe`. Nous pourrons donner un nom à cette classe : `Voiture` (voir figure 20.2).

Vous remarquerez que les classes commencent en général par une majuscule. Visual C# Express nous génère le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MaPremiereApplication
{
    class Voiture
    {
    }
}
```

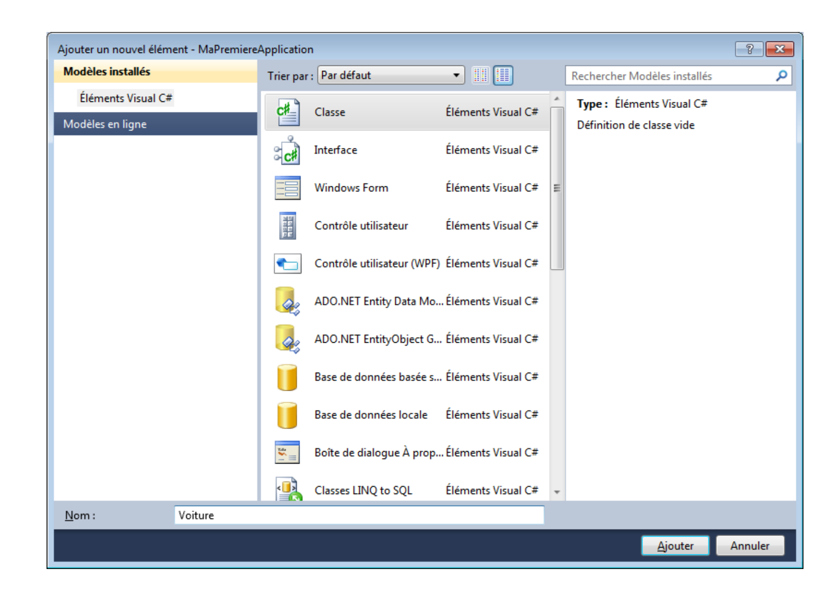


FIGURE 20.2 – Choix d'un modèle de fichier `Classe` dans la fenêtre de sélection des modèles

Nous retrouvons le mot-clé `class` suivi du nom de notre classe `Voiture` et les accolades ouvrantes et fermantes permettant de délimiter notre classe. Notons également que cette classe fait partie de l'espace de nom `MaPremiereApplication` qui est l'espace de nom par défaut de notre projet. Pour nous simplifier le travail, Visual C# Express nous a également inclus quelques espaces de noms souvent utilisés.

Mais pour l'instant cette classe ne fait pas grand-chose.

Comme cette classe est vide, elle ne possède ni propriétés ni actions. Nous ne pouvons absolument rien faire avec, à part en créer une instance, c'est-à-dire un objet. Cela se fait grâce à l'utilisation du mot-clé `new`. Nous y reviendrons plus en détail plus tard, mais cela donne :

```
static void Main(string[] args)
{
    Voiture voitureNicolas = new Voiture();
    Voiture voitureJeremie = new Voiture();
}
```

Nous avons créé deux instances de l'objet `Voiture` et nous les stockons dans les variables `voitureNicolas` et `voitureJeremie`.

Si vous vous rappelez bien, nous aurions logiquement dû écrire :

```
MaPremiereApplication.Voiture voitureNicolas = new
    MaPremiereApplication.Voiture();
```

Ou alors positionner le `using` qui allait bien, permettant d'inclure l'espace de nom `MaPremiereApplication`.

En fait, ici c'est superflu vu que nous créons les objets depuis la méthode `Main()`, qui fait partie de la classe `Program`, faisant partie du même espace de nom que notre classe.

Les méthodes

Nous venons de créer notre objet `Voiture` mais nous ne pouvons pas en faire grand-chose. Ce qui est bien dommage. Ça serait bien que notre voiture puisse klaxonner par exemple si nous sommes bloqués dans des embouteillages. Bref, que notre voiture soit capable de faire des actions.

Qui dit « action » dit « **méthode** ».

Nous allons pouvoir définir des méthodes faisant partie de notre objet `Voiture`. Pour ce faire, il suffit de créer une méthode, comme nous l'avons déjà vu, directement dans le corps de la classe :

```
class Voiture
{
    void Klaxonner()
    {
        Console.WriteLine("Pouet !");
    }
}
```

Notez quand même l'absence du mot-clé `static` que nous étions obligés de mettre avant. Je vous expliquerai un peu plus loin pourquoi.

Ce qui fait que si nous voulons faire klaxonner notre voiture, nous aurons juste besoin d'invoquer la méthode `Klaxonner()` depuis l'objet `Voiture`, ce qui s'écrit :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Klaxonner();
```

Cela ressemble beaucoup à ce que nous avons déjà fait. En fait, nous avons déjà utilisé des méthodes sur des objets. Rappelez-vous, nous avons utilisé la méthode `Add()` permettant d'ajouter une valeur à une liste :

```
List<int> maListe = new List<int>();
maListe.Add(1);
```

Comme nous avons un peu plus de notions désormais, nous pouvons remarquer que nous instancions un objet `List` (plus précisément, une liste d'entier) grâce au mot-clé `new` et que nous invoquons la méthode `Add` de notre liste pour lui ajouter l'entier 1.

Nous avons fait pareil pour obtenir un nombre aléatoire, dans une écriture un peu plus concise. Nous avions écrit :

```
int valeurATrouver = new Random().Next(0, 100);
```

Ce qui peut en fait s'écrire :

```
Random random = new Random();
int valeurATrouver = random.Next(0, 100);
```

Nous créons un objet du type `Random` grâce à `new` puis nous appelons la méthode `Next()` qui prend en paramètres les bornes du nombre aléatoire que nous souhaitons obtenir (0 étant inclus et 100 exclu). Puis nous stockons le résultat dans un entier.

Eh oui, nous avons manipulé quelques objets sans le savoir. . . !

Revenons à notre embouteillage et compilons le code nous permettant de faire klaxonner notre voiture :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Klaxonner();
```

Impossible de compiler, le compilateur nous indique l'erreur suivante :

```
MaPremiereApplication.Voiture.Klaxonner()' est inaccessible en
   raison de son niveau de protection
```

Diantre ! Déjà un premier échec dans notre apprentissage de l'objet !

Vous aurez deviné grâce au message d'erreur que la méthode `Klaxonner()` semble inaccessible. Nous expliquerons un peu plus loin de quoi il s'agit. Pour l'instant, nous allons juste préfixer notre méthode du mot-clé `public`, comme ceci :

```
class Voiture
{
    public void Klaxonner()
    {
        Console.WriteLine("Pouet !");
    }
}
```

Nous allons y revenir juste après, mais le mot-clé `public` permet d'indiquer que la méthode est accessible depuis n'importe où.

Exécutons notre application et nous obtenons :

```
Pouet !
```

Waouh, la première action d'un de nos objets !

Bien sûr, ces méthodes peuvent également avoir des paramètres et renvoyer un résultat. Par exemple :

```
class Voiture
{
    public bool VitesseAutorisee(int vitesse)
    {
        if (vitesse > 90)
            return false;
        else
            return true;
    }
}
```

Cette méthode accepte une vitesse en paramètre et si elle est supérieure à 90, alors la vitesse n'est pas autorisée. Cette méthode pourrait également s'écrire :

```
class Voiture
{
    public bool VitesseAutorisee(int vitesse)
    {
        return vitesse <= 90;
    }
}
```

En effet, nous souhaitons renvoyer faux si la vitesse est supérieure à 90 et vrai si la vitesse est inférieure ou égale.

Donc en fait, nous souhaitons renvoyer la valeur du résultat de la comparaison d'infériorité ou d'égalité de la vitesse à 90, c'est-à-dire :

```
class Voiture
{
    public bool VitesseAutorisee(int vitesse)
    {
        bool estVitesseAutorisee = vitesse <= 90;
        return estVitesseAutorisee;
    }
}
```

Ce que nous pouvons écrire finalement en une seule ligne comme précédemment.

Il est bien sûr possible d'avoir plusieurs méthodes dans une même classe et elles peuvent s'appeler entre elles, par exemple :

```
class Voiture
{
    public bool VitesseAutorisee(int vitesse)
    {
        return vitesse <= 90;
    }

    public void Klaxonner()
    {
        if (!VitesseAutorisee(180))
            Console.WriteLine("Pouet !");
    }
}
```

Quitte à rouler à une vitesse non autorisée, autant faire du bruit !

Notion de visibilité



Oui, mais le mot-clé `public`... il nous a bien sauvé la vie, mais... qu'est-ce donc ?

En fait, je l'ai rapidement évoqué et nous nous sommes bien rendu compte que sans ce mot-clé, impossible de compiler car la méthode n'était pas accessible.

Le mot-clé `public` sert à indiquer que notre méthode pouvait être accessible depuis d'autres classes ; en l'occurrence dans notre exemple depuis la classe `Program`. C'est-à-dire que sans ce mot-clé, il est impossible à d'autres objets d'utiliser cette méthode.

Pour faire en sorte qu'une méthode soit inaccessible, nous pouvons utiliser le mot-clé `private`. Ce mot-clé permet d'avoir une méthode qui n'est accessible que depuis la classe dans laquelle elle est définie. Prenons l'exemple suivant :

```
class Voiture
{
    public bool Demarrer()
    {
        if (ClesSurLeContact())
        {
            DemarrerLeMoteur();
            return true;
        }
        return false;
    }

    public void SortirDeLaVoiture()
    {
        if (ClesSurLeContact())
            PrevenirLUtilisateur();
    }

    private bool ClesSurLeContact()
    {
        // faire quelque chose pour vérifier
        return true;
    }

    private void DemarrerLeMoteur()
    {
        // faire quelque chose pour démarrer le moteur
    }

    private void PrevenirLUtilisateur()
    {
        Console.WriteLine("Bip bip bip");
    }
```

```
34 }
```

Ici seules les méthodes `Demarrer()` et `SortirDeLaVoiture()` sont utilisables depuis une autre classe, c'est-à-dire que ce sont les seules méthodes que nous pourrons invoquer, car elles sont publiques. Les autres méthodes sont privées à la classe et ne pourront être utilisées qu'à l'intérieur de la classe elle-même. Les autres classes n'ont pas besoin de savoir comment démarrer le moteur ou comment vérifier que les clés sont sur le contact, elles n'ont besoin que de pouvoir démarrer ou sortir de la voiture. Les méthodes privées sont exclusivement réservées à l'usage interne de la classe.



Notez d'ailleurs que la complétion automatique n'est pas proposée pour les méthodes inaccessibles.

Il existe d'autres indicateurs de visibilités que nous allons rapidement décrire :

Visibilité	Description
`public`	Accès non restreint
`protected`	Accès depuis la même classe ou depuis une classe dérivée
`private`	Accès uniquement depuis la même classe
`internal`	Accès restreint à la même assembly
`protected internal`	Accès restreint à la même assembly ou depuis une classe dérivée

Les visibilités qui vont le plus vous servir sont représentées par les mots clés `public` et `private`. Nous verrons que le mot-clé `protected` va servir un peu plus tard quand nous parlerons d'héritage. Notez qu'`internal` pourra être utilisé une fois que nous aurons bien maîtrisé toutes les notions.

Ces mots-clés sont utilisables avec beaucoup d'autres concepts. Nous avons utilisé les méthodes pour les illustrer mais ceci est également valable pour les classes ou les propriétés que nous allons découvrir juste après.



Au début, nous avons pu déclarer une classe sans préciser de visibilité. . . et pareil pour la première méthode qui ne compilait pas. . . c'est normal ?

Oui, il existe des visibilités par défaut suivant les types déclarés. Vous aurez compris par exemple que la visibilité par défaut d'une méthode est privée si l'on ne spécifie pas le mot-clé.

Pour éviter tout risque et toute ambiguïté, il est recommandé de toujours indiquer la visibilité ; ce que nous ferons désormais dans ce livre, maintenant que nous savons de quoi il s'agit.

Les propriétés

Des objets c'est bien. Des actions sur ces objets, c'est encore mieux. Il nous manque encore les caractéristiques des objets. C'est là qu'interviennent les propriétés.

Sans le savoir, vous avez déjà utilisé des propriétés, par exemple dans le code suivant :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
for (int i = 0; i < jours.Length; i++)
{
    Console.WriteLine(jours[i]);
}
```

Dans la boucle, nous utilisons `jours.Length`. Nous utilisons en fait la propriété `Length` du tableau « jours », un tableau étant bien sûr un objet.

Nous avons pu utiliser d'autres propriétés, par exemple dans l'instruction suivante :

```
List<string> jours = new List<string> { "Lundi", "Mardi", "
    Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
for (int i = 0; i < jours.Count; i++)
{
    Console.WriteLine(jours[i]);
}
```

Ici, `Count` est une propriété de la liste « jours ».

De la même façon, nous avons la possibilité de créer des propriétés sur nos classes pour permettre d'ajouter des caractéristiques à nos objets.

Par exemple, nous pouvons rajouter les propriétés suivantes à notre voiture : une couleur, une marque, une vitesse. Il y a plusieurs façons de rajouter des caractéristiques à un objet. La première est d'utiliser des variables membres.

Les variables membres

Ici en fait, un objet peut avoir une caractéristique sous la forme d'une variable publique qui fait partie de la classe. Pour ce faire, il suffit de définir simplement la variable à l'intérieur des blocs de code délimitant la classe et de lui donner la visibilité `public`.

Rajoutons par exemple une chaîne de caractères permettant de stocker la couleur de la voiture :

```
public class Voiture
{
    public string Couleur;
}
```

Notez que j'ai rajouté les visibilités pour la classe et pour la variable.

Grâce à ce code, la chaîne de caractères `Couleur` est désormais une caractéristique de la classe `Voiture`. Nous pourrons l'utiliser en faisant suivre notre objet de l'opérateur

« . » suivi du nom de la variable. Ce qui donne :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Couleur = "rouge";
Voiture voitureJeremie = new Voiture();
voitureJeremie.Couleur = "verte";
```

Cela ressemble beaucoup à ce que nous avons déjà fait. Nous utilisons le « . » pour accéder aux propriétés d'un objet. Comme d'habitude, les variables vont pouvoir stocker des valeurs. Ainsi, nous pourrons avoir une voiture rouge pour Nicolas et une voiture verte pour Jérémie.



Notez ici qu'il s'agit bien de variables membres et non de vraies propriétés. En général, les variables d'une classe ne doivent jamais être publiques. Nous utiliserons rarement cette construction.

Les propriétés :

Les propriétés sont en fait des variables évoluées. Elles sont à mi-chemin entre une variable et une méthode.

Prenons cet exemple :

```
public class Voiture
{
    private int vitessePrivee;
    public int Vitesse
    {
        get
        {
            return vitessePrivee;
        }
        set
        {
            vitessePrivee = value;
        }
    }
}
```

Nous pouvons voir que nous définissons dans un premier temps une variable privée, `vitessePrivee` de type entier. Comme prévu, cette variable est masquée pour les utilisateurs d'objets `Voiture`. Ainsi, le code suivant :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.vitessePrivee = 50;
```

provoquera l'erreur de compilation désormais bien connue :

```
MaPremiereApplication.Voiture.vitessePrivee' est inaccessible en
    raison de son niveau de protection
```

Par contre, nous en avons profité pour définir la propriété `Vitesse`, de type `int`. Pour ceci, nous avons défini une partie de la propriété avec le mot-clé `get` suivi d'un `return vitessePrivee`. Et juste en dessous, nous avons utilisé le mot-clé `set`, suivi de `vitessePrivee = value`.

Ce que nous avons fait, c'est définir la possibilité de lire la propriété grâce au mot-clé `get`. Ici, la lecture de la propriété nous renvoie la valeur de la variable privée. De la même façon, nous avons défini la possibilité d'affecter une valeur à la propriété en utilisant le mot-clé `set` et en affectant la valeur à la variable privée.

Les blocs de code délimités par `get` et `set` se comportent un peu comme des méthodes. Ils ont un corps qui est délimité par des accolades et dans le cas du `get`, ils doivent renvoyer une valeur du même type que la propriété.

À noter que dans le cas du `set`, « `value` » est un mot-clé qui permet de dire : « la valeur que nous avons affectée à la propriété ».

Prenons l'exemple suivant :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Vitesse = 50;
Console.WriteLine(voitureNicolas.Vitesse);
```

La première instruction instancie bien sûr une voiture.

La deuxième instruction consiste à appeler le bloc de code `set` de `Vitesse` qui met la valeur 50 dans la pseudo-variable `value` qui est stockée ensuite dans la variable privée.

Lorsque nous appelons la troisième instruction, nous lisons la valeur de la propriété et pour ce faire, nous passons par le `get` qui nous renvoie la valeur de la variable privée.



Ok, c'est super, mais dans ce cas, pourquoi passer par une propriété et pas par une variable ? Même s'il paraît que les variables ne doivent jamais être publiques. . .

Eh bien parce que dans ce cas-là, la propriété peut faire un peu plus que simplement renvoyer une valeur. Et aussi parce que nous masquons la structure interne de notre classe à ceux qui veulent l'utiliser. Nous pourrions très bien envisager d'aller lire la vitesse dans les structures internes du moteur, ou en faisant un calcul poussé par rapport au coefficient du vent et de l'âge du capitaine (ou plus vraisemblablement en allant lire la valeur en base de données). Et ici, nous pouvons tirer parti de la puissance des propriétés pour masquer tout ça à l'appelant qui, lui, n'a besoin que d'une vitesse.

Par exemple :

```
private int vitessePrivee;
public int Vitesse
{
    get
    {
        int v = vitesseDesRoues * rayon * coefficient; // ce
            calcul est complètement farfelu !
        MettreAJourLeCompteur();
```

```
            AdapterLaVitesseDesEssuieGlaces();
            return v;
        }
        set
        {
            // faire la mise à jour des variables internes
            MettreAJourLeCompteur();
            AdapterLaVitesseDesEssuieGlaces();
        }
}
```

En faisant tout ça dans le bloc de code `get`, nous masquons les rouages de notre classe à l'utilisateur. Lui, il n'a besoin que d'obtenir la vitesse, sans s'encombrer du compteur ou des essuie-glaces.

Bien sûr, la même logique peut s'adapter au bloc de code `set`, qui permet d'affecter une valeur à la propriété.

Il est également possible de rendre une propriété en lecture seule, c'est-à-dire non modifiable par les autres objets. Il pourra par exemple sembler bizarre de positionner une valeur à la vitesse alors qu'en fait, pour mettre à jour la vitesse, il faut appeler la méthode `AppuyerPedale(double forceAppui)`.

Pour empêcher les autres objets de pouvoir directement mettre à jour la propriété `Vitesse`, il suffit de ne pas déclarer le bloc de code `set` et de ne garder qu'un `get`. Par exemple :

```
public class Voiture
{
    private int vitessePrivee;
    public int Vitesse
    {
        get
        {
            // faire des calculs ...
            return vitessePrivee;
        }
    }
}
```

Ce qui fait que si nous tentons d'affecter une valeur à la propriété `Vitesse` depuis une autre classe, par exemple :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Vitesse = 50;
```

Nous aurons l'erreur de compilation suivante :

```
La propriété ou l'indexeur 'MaPremiereApplication.Voiture.
   Vitesse' ne peut pas être assigné -- il est en lecture seule
```

Le compilateur nous indique donc très justement qu'il est impossible de faire cette affectation car la propriété est en lecture seule. Il devient donc impossible pour un utilisateur de cette classe de modifier la vitesse de cette façon.

De même, il est possible de définir une propriété pour qu'elle soit accessible en écriture seule. Il suffit de définir uniquement le bloc de code `set` :

```
private double acceleration;
public double Acceleration
{
    set
    {
        acceleration = value;
    }
}
```

Ainsi, si nous tentons d'utiliser la propriété en lecture, avec par exemple :

```
Console.WriteLine(voitureNicolas.Acceleration);
```

Nous aurons l'erreur de compilation attendue :

```
La propriété ou l'indexeur 'MaPremiereApplication.Voiture.
    Acceleration' ne peut pas être utilisé dans ce contexte, car
    il lui manque l'accesseur get
```

Ce bridage d'accès à des propriétés prend tout son sens quand nous souhaitons exposer nos objets à d'autres consommateurs qui n'ont aucun intérêt à connaître la structure interne de notre classe. C'est un des principes de l'encapsulation.

Les propriétés auto-implémentées

Les propriétés auto-implémentées sont une fonctionnalité que nous allons beaucoup utiliser. Il s'agit de la définition d'une propriété de manière très simplifiée qui va nous servir dans la grande majorité des cas où nous aurons besoin d'écrire des propriétés. Dans cet ouvrage, nous l'utiliserons très souvent.

Ainsi, le code suivant que nous avons déjà vu :

```
private int vitesse;
public int Vitesse
{
    get
    {
        return vitesse;
    }
    set
    {
        vitesse = value;
    }
}
```

est un cas très fréquent d'utilisation de propriétés. Nous exposons ici une variable privée à travers les propriétés `get` et `set`. L'écriture du dessus est équivalente à la suivante :

```
public int Vitesse { get; set; }
```

Dans ce cas, le compilateur fait le boulot lui-même, il génère (dans le code compilé) une variable membre qui va servir à stocker la valeur de la propriété.

C'est exactement pareil, sauf que cela nous simplifie grandement l'écriture.

En plus, nous pouvons encore accélérer son écriture en utilisant ce qu'on appelle des « **snippets** » qui sont des extraits de code. Pour les utiliser, il suffit de commencer à écrire un mot et Visual C# nous complète le reste. Un peu comme la complétion automatique sauf que cela fonctionne pour des bouts de code très répétitifs et très classiques à écrire.

Commencez par exemple à taper « prop » ; Visual C# nous propose plusieurs extraits de code, comme vous pouvez le voir sur la figure 20.3.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MaPremiereApplication
{
    public class Voiture
    {
        prop
    }
}
```

prop / propa / propdp / propfull / propg

prop
Extrait de code pour une propriété implémentée automatiquement
Version de langage : C# 3.0 ou supérieure

FIGURE 20.3 – Utilisation du snippet de création de propriété auto-implémentée

Appuyez sur Tab ou Entrée pour sélectionner cet extrait de code et appuyez ensuite sur Tab pour que Visual C# génère l'extrait de code correspondant à la propriété auto-implémentée (voir figure 20.4).

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MaPremiereApplication
{
    public class Voiture
    {
        public int MyProperty { get; set; }
    }
}
```

FIGURE 20.4 – Changer le type de la propriété

Ici, `int` est surligné et vous pouvez, si vous le souhaitez, changer le type de la propriété, par exemple `string`. Appuyez à nouveau sur Tab et Visual C# surligne le nom de

la propriété, que vous pouvez à nouveau changer. Appuyez enfin sur Entrée pour terminer la saisie et vous aurez une belle propriété auto-implémentée (voir figure 20.5).

```
using System.Text;

namespace MaPremiereApplication
{
    public class Voiture
    {
        public string Couleur { get; set; }
    }
}
```

FIGURE 20.5 – La propriété auto-implémentée

Vous avez pu remarquer qu'en commençant à taper « `prop` », Visual C# Express vous a proposé d'autres extraits de code, par exemple « `propfull` » qui va générer la propriété complète telle qu'on l'a vue un peu plus haut.

D'autres extraits de code existent, comme « `propg` » qui permet de définir une propriété en lecture seule.

En effet, comme au chapitre précédent, il est possible de définir des propriétés auto-implémentées en lecture seule ou en écriture seule avec une écriture simplifiée. Dans le cas des propriétés auto-implémentées, il y a cependant une subtilité.

Pour avoir de la lecture seule, nous devons indiquer que l'affectation est privée, comme on peut le voir en utilisant l'extrait de code « `propg` ». Visual C# Express nous génère le bout de code suivant :

```
public int Vitesse { get; private set; }
```

En positionnant une propriété d'écriture en privé, Visual C# Express autorise la classe `Voiture` à modifier la valeur de `Vitesse`, que ce soit par une méthode ou par une propriété.

À noter que si nous n'avions pas mis `private set` et que nous avions simplement supprimé le `set`, alors la compilation aurait été impossible. En effet, il s'avère difficile d'exploiter une propriété auto-implémentée en lecture alors que nous n'avons pas la possibilité de lui donner une valeur. Ou inversement.



Alors, pourquoi nous avoir parlé de la possibilité de complètement supprimer la lecture ou l'écriture ? Ce n'est pas plus intéressant de toujours mettre la propriété en `private` ?

Si c'est une propriété auto-implémentée, évidemment que si. Par contre, si nous n'utilisons pas une propriété auto-implémentée et que nous utilisons une variable membre pour sauvegarder la valeur de la propriété, nous pourrons éventuellement modifier ou lire la valeur de la variable à partir d'une méthode ou d'une autre propriété.

Bref, maintenant que vous connaissez les deux syntaxes, vous pourrez utiliser la plus adaptée à votre besoin.

Voilà pour les propriétés.

À noter que quand nous avons beaucoup de propriétés à initialiser sur un objet, nous pouvons utiliser une syntaxe plus concise. Par exemple, les instructions suivantes :

```
Voiture voitureNicolas = new Voiture();

voitureNicolas.Couleur = "Bleue";
voitureNicolas.Marque = "Peugeot";
voitureNicolas.Vitesse = 50;
```

sont équivalentes à l'instruction :

```
Voiture voitureNicolas = new Voiture { Couleur = "Bleue",
    Marque = "Peugeot", Vitesse = 50 };
```

Les accolades servent ici à initialiser les propriétés au même moment que l'instanciation de l'objet.



Note : cela paraît évident, mais il est bien sûr possible d'accéder aux propriétés d'une classe depuis une méthode de la même classe.

En résumé

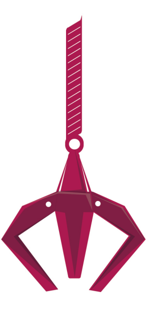
- On utilise des classes pour représenter quasiment la plupart des objets.
- On utilise le mot-clé `class` pour définir une classe et le mot-clé `new` pour l'instancier.
- Une classe peut posséder des caractéristiques (les propriétés) et peut faire des actions (les méthodes).

Chapitre 21

Manipuler des objets

Difficulté : 

Maintenant que nous avons bien vu comment définir des objets, il est temps de savoir les utiliser. Nous allons voir dans ce chapitre quelles sont les subtilités de l'instanciation des objets. Vous verrez notamment ce qu'est un constructeur et qu'on peut avoir des valeurs nulles pour des objets. N'hésitez pas à jouer avec tous ces concepts, il est important d'être à l'aise avec les bases pour pouvoir être efficace avec les concepts avancés !



Le constructeur

Le constructeur est une méthode particulière de l'objet qui permet de faire des choses au moment de la création d'un objet, c'est-à-dire au moment où nous utilisons le mot-clé `new`. Il est en général utilisé pour initialiser des valeurs par défaut d'un objet. Par exemple, si nous voulons que lors de la création d'une voiture, elle ait automatiquement une vitesse, nous pouvons faire :

```
public class Voiture
{
    public int Vitesse { get; set; }

    public Voiture()
    {
        Vitesse = 5;
    }
}
```

Le constructeur est en fait une méthode spéciale qui a le même nom que la classe et qui ne possède pas de type de retour. Elle est appelée lors de la création de l'objet, avec `new`.

Pour illustrer le comportement du constructeur, ajoutons une méthode `Rouler` à notre classe, de cette façon :

```
public class Voiture
{
    public int Vitesse { get; set; }

    public Voiture()
    {
        Vitesse = 5;
    }

    public void Rouler()
    {
        Console.WriteLine("Je roule à " + Vitesse + " km/h");
    }
}
```

Que nous pourrons appeler ainsi :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Rouler();
voitureNicolas.Vitesse = 50;
voitureNicolas.Rouler();
```

Au moment de l'instanciation de l'objet avec `new`, la vitesse va être égale à 5. Nous faisons rouler la voiture. Puis nous changeons la vitesse pour la passer à 50 et nous faisons à nouveau rouler la voiture :

```
Je roule à 5 km/h
Je roule à 50 km/h
```

Le constructeur est la première « méthode » à être appelée lors de la création d'un objet. C'est l'endroit approprié pour faire des initialisations, ou pour charger des valeurs, etc.

Le constructeur que nous avons vu est ce qu'on appelle **le constructeur par défaut**, car il ne possède pas de paramètres. Il est possible de passer des paramètres à un constructeur, pour initialiser des variables de notre classe avec des valeurs. Pour ce faire, nous devons déclarer un constructeur avec un paramètre. Par exemple :

```
public class Voiture
{
    public int Vitesse { get; set; }

    public Voiture()
    {
        Vitesse = 5;
    }

    public Voiture(int vitesse)
    {
        Vitesse = vitesse;
    }

    public void Rouler()
    {
        Console.WriteLine("Je roule à " + Vitesse + " km/h");
    }
}
```

Ainsi, nous pourrons créer un objet voiture en lui précisant une vitesse par défaut de cette façon :

```
Voiture voitureNicolas = new Voiture(20);
```

Après ceci, la variable `voitureNicolas` aura une vitesse de 20.

Bien sûr, nous pourrions faire la même chose sans utiliser de constructeur, en affectant une valeur à la propriété après avoir instancié l'objet. Ce qui fonctionnerait tout à fait. Sauf qu'il ne se passe pas exactement la même chose. Le constructeur est vraiment appelé en premier, dès qu'on utilise `new` pour créer un objet. Les propriétés sont affectées fatalement après, donc tout dépend de ce que l'on veut faire.

Donc, oui, on pourrait affecter une valeur à des propriétés pour faire ce genre d'initialisation juste après avoir instancié notre objet mais cela nous oblige à écrire une instruction supplémentaire qui pourrait ne pas paraître évidente ou obligatoire.

Une chose est sûre avec le constructeur, c'est que nous sommes obligés d'y passer et ceci, peu importe la façon dont on utilise la classe. L'initialisation devient donc obligatoire, on évite le risque qu'une propriété soit nulle.

À noter qu'il est possible de cumuler les constructeurs tant qu'ils ont chacun des paramètres différents. Dans notre exemple, nous pourrons donc créer des voitures de deux façons différentes :

```
Voiture voitureNicolas = new Voiture(); // vitesse vaut 5
Voiture voitureJeremie = new Voiture(20); // vitesse vaut 20
```

Il est aussi possible de ne pas définir de constructeur par défaut et d'avoir uniquement un constructeur possédant des paramètres. Dans ce cas, il devient impossible d'instancier un objet sans lui passer de paramètres.

Instancier un objet

Nous allons revenir à présent sur l'instanciation d'un objet. Comme nous venons de le voir, nous utilisons le mot-clé `new` pour créer une instance d'un objet. C'est lui qui permet la création d'un objet. Il appelle le constructeur correspondant. Si aucun constructeur n'existe, il ne se passera rien de plus qu'une création de base. Par exemple :

```
Voiture voitureNicolas = new Voiture();
```

Pour rentrer un peu dans la technique, au moment de l'instanciation d'un objet, l'opérateur `new` crée l'objet et le met à une place disponible en mémoire. Cette adresse mémoire est conservée dans la variable `voitureNicolas`. On dit que `voitureNicolas` contient une **référence** vers l'objet. Nous verrons un peu plus tard ce que cela implique.



Ce principe ressemble un peu au « pointeur » que nous pourrions trouver dans des langages comme le C ou le C++. Typiquement, si vous savez ce qu'est un pointeur, vous pouvez vous représenter une référence comme un pointeur évolué.

Comme pour les types que nous avons vus plus haut, nous sommes obligés d'initialiser un objet avant de l'utiliser. Sinon, Visual C# Express nous générera une erreur de compilation. Par exemple, les instructions suivantes :

```
Voiture voitureNicolas;
voitureNicolas.Vitesse = 5;
```

provoqueront l'erreur de compilation suivante :

```
Utilisation d'une variable locale non assignée 'voitureNicolas'
```

En effet, Visual C# Express est assez intelligent pour se rendre compte que l'on va essayer d'accéder à un objet qui n'a pas été initialisé.

Il faut toujours initialiser une variable avant de pouvoir l'utiliser. Comme pour les types précédents, il est possible de dissocier la déclaration d'un objet de son instanciation, en écrivant les instructions sur plusieurs lignes, par exemple :

```
Voiture voitureNicolas;
// des choses
voitureNicolas = new Voiture();
```

ceci est possible tant que nous n'utilisons pas la variable `voitureNicolas` avant de l'avoir instanciée.

Les objets peuvent également avoir une valeur nulle. Ceci est différent de l'absence d'initialisation car la variable est bien initialisée et sa valeur vaut « nul ». Ceci est possible grâce à l'emploi du mot-clé `null`.

```
Voiture voitureNicolas = null;
```

Attention, il est par contre impossible d'accéder à un objet qui vaut `null`. Eh oui, comment voulez-vous vous asseoir sur une chaise qui n'existe pas ? Eh bien vous vous retrouverez avec les fesses par terre, personne ne vous a indiqué que la chaise n'existait pas.

C'est pareil pour notre application, si nous tentons d'utiliser une voiture qui n'existe pas, nous aurons droit à un beau plantage.

Par exemple, avec le code suivant :

```
Voiture voitureNicolas = null;
voitureNicolas.Vitesse = 5;
```

vous n'aurez pas d'erreur à la compilation, par contre vous aurez un message d'exception assez explicite :

```
Exception non gérée : System.NullReferenceException: La réfé
    rence d'objet n'est pas définie à une instance d'un objet.
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    13
```

Comme nous l'avons déjà vu, le programme nous affiche une exception ; nous avons dit que c'était simplement une erreur qui faisait planter notre programme. Ce n'est pas bien ! Surtout que cela se passe au moment de l'exécution. Nous perdons toute crédibilité !

Ici, le programme nous dit que la référence d'un objet n'est pas définie à une instance d'un objet. Concrètement, cela veut dire que nous essayons de travailler sur un objet `null`.

Pour éviter ce genre d'erreur à l'exécution, il faut impérativement instancier ses objets, en utilisant l'opérateur `new`, comme nous l'avons déjà vu. Il n'est cependant pas toujours pertinent d'instancier un objet dont on pourrait ne pas avoir besoin. Le C# nous offre donc la possibilité de tester la nullité d'un objet. Il suffit d'utiliser l'opérateur de comparaison « == » en comparant un objet au mot-clé `null`, par exemple :

```
string prenom = "Nicolas";
Voiture voiture = null;
```

```
if (prenom == "Nicolas")
    voiture = new Voiture { Vitesse = 50 };
if (voiture == null)
{
    Console.WriteLine("Vous n'avez pas de voiture");
}
else
{
    voiture.Rouler();
}
```

Ainsi, seul `Nicolas` possédera une voiture et le test de nullité sur l'objet permet d'éviter une erreur d'exécution si le prénom est différent.

Maintenant que vous connaissez le mot-clé `null` et que vous savez qu'un objet peut prendre une valeur nulle, nous allons revenir sur un point que j'ai rapidement abordé auparavant. Je ne sais pas si vous vous en rappelez, mais lors de l'étude des opérateurs logiques j'ai parlé du fait que l'opérateur OU (||) évaluait la première condition et si elle était vraie alors il n'évaluait pas la suivante, considérant que de toute façon, le résultat allait être vrai.

Ce détail prend toute son importance dans le cas suivant :

```
if (voiture == null || voiture.Couleur == "Bleue")
{
    // faire quelque chose
}
```

Dans ce cas, si la voiture est effectivement nulle, alors le fait d'évaluer la propriété `Couleur` de la voiture devrait renvoyer une erreur. Heureusement, le C# avait prévu le coup. Si la première condition est vraie alors la seconde ne sera pas évaluée, ce qui évitera l'erreur. Ainsi, nous sommes sûrs de n'avoir aucune voiture bleue.

Il est par contre évident qu'une telle condition utilisant l'opérateur ET (&&) est une hérésie car pour que la condition soit vraie, le C# a besoin d'évaluer les deux opérandes. Et donc si la voiture est nulle, l'utilisation d'une propriété sur une valeur nulle renverra une erreur.

Notons également que lorsque nous utilisons l'opérateur ET (&&), si la première opérande est fausse, alors de la même façon, il n'évalue pas la seconde, car pour que la condition soit vraie il faut que les deux le soient. Ce qui fait qu'il est également possible d'écrire ce code :

```
if (voiture != null && voiture.Couleur == "Rouge")
{
        // faire autre chose
}
```

qui ne provoquera pas d'erreur à l'exécution, même si voiture vaut `null` car dans ce cas, le fait que le premier test soit faux évitera le test de l'autre partie de l'expression.

Vous verrez que vous aurez l'occasion d'utiliser le mot-clé `null` régulièrement.

Le mot-clé `this`

Lorsque nous écrivons le code d'une classe, le mot-clé `this` représente l'objet dans lequel nous nous trouvons. Il permet de clarifier éventuellement le code, mais il est généralement facultatif. Ainsi, pour accéder à une variable de la classe ou éventuellement une méthode, nous pouvons les préfixer par « `this.` ». Par exemple, nous pourrions écrire notre classe de cette façon :

```
public class Voiture
{
    public int Vitesse { get; set; }
    public string Couleur { get; set; }

    public Voiture()
    {
        this.Vitesse = 5;
    }

    public void Rouler()
    {
        Console.WriteLine("Je roule à " + this.Vitesse + " km/h");
    }

    public void Accelerer(int acceleration)
    {
        this.Vitesse += acceleration;
        this.Rouler();
    }
}
```

Ici, dans le constructeur, nous utilisons le mot-clé `this` pour accéder à la propriété `Vitesse`. C'est la même chose dans la méthode `Rouler`. De la même façon, on peut utiliser `this.Rouler()` pour appeler la méthode `Rouler` depuis la méthode `Accelerer()`.

C'est une façon pour la classe de dire : « Regardez, avec `this`, c'est "ma variable à moi" ».

Notez bien sûr que sans le mot-clé `this`, notre classe compilera quand même et sera tout à fait fonctionnelle. Ce mot-clé est facultatif mais il peut aider à bien faire la différence entre ce qui appartient à la classe et ce qui fait partie des paramètres des méthodes ou d'autres objets utilisés.

Suivant les personnes, le mot-clé `this` est soit systématiquement utilisé, soit jamais. Je fais plutôt partie des personnes qui ne l'utilisent jamais. Il arrive par contre certaines situations où il est absolument indispensable, comme celle-ci, mais en général, j'essaie d'éviter ce genre de construction :

```
public void ChangerVitesse(int Vitesse)
{
    this.Vitesse = Vitesse;
```

```
}
```

Vous remarquerez que le paramètre de la méthode `ChangerVitesse()` et la propriété ou variable membre de la classe ont exactement le même nom. Ceci est possible ici mais source d'erreurs, les variables ayant des portées différentes. Il s'avère que dans ce cas, le mot-clé `this` est indispensable. On pourra donc éviter l'ambiguïté en préfixant la propriété membre avec le mot-clé `this`.

Je recommande plutôt de changer le nom du paramètre, quitte à utiliser une minuscule, ce qui augmentera la lisibilité et évitera des erreurs potentielles.

En général, des conventions de nommage pourront nous éviter de nous retrouver dans ce genre de situation.

Ça y est, nous savons instancier et utiliser des objets ! Savoir créer et utiliser ses propres objets est très important dans un programme orienté objet. Vous allez également avoir besoin très régulièrement d'utiliser des objets tout faits, comme ceux venant de la bibliothèque de classe du framework .NET. Nous comprenons d'ailleurs mieux pourquoi elle s'appelle « bibliothèque de classes ». Il s'agit bien d'un ensemble de classes utilisables dans notre application et nous pourrons instancier les objets relatifs à ces classes pour nos besoins. Comme ce que nous avions déjà fait auparavant sans trop le savoir, avec l'objet `Random` par exemple. . .

N'hésitez pas à relire ce chapitre ainsi que le précédent si vous n'avez pas parfaitement compris toutes les subtilités de la création d'objet. C'est un point important du livre.

En résumé

– Les classes possèdent une méthode particulière, appelée à l'instanciation de l'objet : le constructeur.
– Une instance d'une classe peut être initialisée avec une valeur nulle grâce au mot-clé `null`.
– Le mot-clé `this` représente l'objet en cours de la classe.

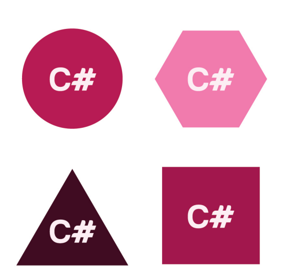
Chapitre 22

La POO et le C#

Difficulté : 

Dans ce chapitre, vous allez vous immerger un peu plus dans les subtilités de la POO en utilisant le C#. Il est temps un peu de tourmenter nos objets et de voir ce qu'ils ont dans le ventre. Ainsi, nous allons voir comment les objets héritent les uns des autres ou comment fonctionnent les différents polymorphismes.

Nous allons également voir comment tous ces concepts se retrouvent dans le quotidien d'un développeur C#.



Des types, des objets, type valeur et type référence



Ok, je sais maintenant créer des objets, mais je me rappelle qu'au début du livre, nous avons manipulé des `int` et des `string` et que tu as appelé ça des « types » ; et après, tu nous dis que tout est objet... Tu serais pas en train de raconter n'importe quoi par hasard ?!

Eh bien non, ô perspicace lecteur ! Précisons un peu, maintenant que vous avez de meilleures connaissances. J'ai bien dit que tout était objet, je le maintiens, même sous la torture ! C'est-à-dire que même les types simples comme les entiers `int` ou les chaînes de caractères sont des objets.

J'en veux pour preuve ce simple exemple :

```
int a = 10;
string chaine = a.ToString();
chaine = "abc" + chaine;
string chaineEnMajuscule = chaine.ToUpper();
Console.WriteLine(chaineEnMajuscule);
Console.WriteLine(chaineEnMajuscule.Length);
```

La variable `a` est un entier. Nous appelons la méthode `ToString()` sur cet entier. Même si nous n'avons pas encore vu à quoi elle servait, nous pouvons supposer qu'elle effectue une action qui consiste à transformer l'entier en chaîne de caractères. Nous concaténons ensuite la chaîne `abc` à cette chaîne et nous effectuons une action qui, à travers la méthode `ToUpper()`, met la chaîne en majuscule. Enfin, la méthode `Console.WriteLine` nous affiche « `ABC10` » puis nous affiche la propriété `Length` de la chaîne de caractères qui correspond bien sûr à sa taille.

Pour créer une chaîne de caractères, nous utilisons le mot-clé `string`. Sachez que ce mot-clé est équivalent à la classe `String` (notez la différence de casse). En créant une chaîne de caractères, nous avons instancié un objet défini par la classe `String`.



Mais alors, pourquoi utiliser `string` et non pas `String` ?

En fait, le mot-clé `string` est ce qu'on appelle un alias de la classe `String` qui se situe dans l'espace de nom `System`. De même, le mot-clé `int` est un alias de la structure `Int32` qui se situe également dans l'espace de nom `System` (nous verrons un peu plus loin ce qu'est vraiment une structure).

Ce qui fait que les instructions suivantes :

```
int a = 10;
string chaine = "abc";
```

sont équivalentes à celles-ci :

```
System.Int32 a = 10;
```

```
System.String chaine = "abc";
```



En pratique, comme on l'a déjà fait, on utilise plutôt les alias que les classes qu'ils représentent.

Cependant, les entiers, les booléens et autres types « simples » sont ce qu'on appelle des types intégrés. Et même si ce sont des objets à part entière (méthodes, propriétés,...), ils ont des particularités, notamment dans la façon dont ils sont gérés par le framework .NET. On les appelle des **types valeur**, car les variables de ce type possèdent la vraie valeur de ce qu'on leur affecte *a contrario* des classes qui sont des **types référence** dont les variables possèdent simplement un lien vers un objet en mémoire.

Par exemple :

```
int entier = 5;
```

Ici, la variable contient vraiment l'entier 5. Alors que pour l'instanciation suivante :

```
Voiture voitureNicolas = new Voiture();
```

La variable `voitureNicolas` contient une référence vers l'objet en mémoire.

On peut imaginer que le type référence est un peu comme si on disait que ma maison se situe au « 9 rue des bois ». L'adresse a été écrite sur un bout de papier et référence ma maison qui ne se situe bien sûr pas au même endroit que le bout de papier. Si je veux vraiment voir l'objet maison, il va falloir que j'aille voir où c'est indiqué sur le bout de papier. C'est ce que fait le type référence, il va voir en mémoire ce qu'il y a vraiment dans l'objet.

Alors que le type valeur pourrait ressembler à un billet de banque par exemple. Je peux me balader avec, il est marqué 500 € dessus (oui, je suis riche !) et je peux payer directement avec sans que le fait de donner le billet implique d'aller chercher le contenu à la banque.

- Le **type valeur** contient la vraie valeur qui en général est assez petite et facile à stocker.
- Le **type référence** ne contient qu'un lien vers un plus gros objet stocké ailleurs.

Cette manière différente de gérer les types et les objets implique plusieurs choses. Dans la mesure où les types valeur possèdent vraiment la valeur de ce qu'on y stocke, une copie de la valeur est effectuée à chaque fois que l'on fait une affectation. C'est possible car ces types sont relativement petits et optimisés. Cela s'avère impossible pour un objet qui est trop gros et trop complexe. C'est un peu compliqué de copier toute ma maison alors que c'est un peu plus simple de recopier ce qu'il y a sur le bout de papier.

Ainsi, l'exemple suivant :

```
int a = 5;
int b = a;
b = 6;
Console.WriteLine(a);
```

```
Console.WriteLine(b);
```

affichera les valeurs 5 puis 6 ; ce qui est le résultat que l'on attend.

– En effet, la variable `a` a été initialisée à 5.
– On a ensuite affecté `a` à `b`. La valeur 5 s'est copiée (dupliquée) dans la variable `b`.
– Puis nous avons affecté 6 à `b`.

Ce qui paraît tout à fait logique ! Par contre, l'exemple suivant :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Couleur = "Bleue";
Voiture voitureJeremie = voitureNicolas;
voitureJeremie.Couleur = "Verte";
Console.WriteLine(voitureNicolas.Couleur);
Console.WriteLine(voitureJeremie.Couleur);
```

affichera verte et verte.



Quoi ? Nous indiquons que la voiture de Nicolas est bleue. Puis nous disons que celle de Jérémie est verte et quand on demande d'afficher la couleur des deux voitures, on nous dit qu'elles sont vertes toutes les deux alors qu'on croyait que celle de Nicolas était bleue ? Tout à l'heure, le fait de changer `b` n'avait pas changé la valeur de a. . .

Eh oui, ceci illustre le fait que les classes (comme `Voiture`) sont des types référence et ne possèdent qu'une référence vers une instance de `Voiture`. Quand nous affectons `voitureNicolas` à `voitureJeremie`, nous disons en fait que la voiture de Jérémie référence la même chose que celle de Nicolas. Concrètement, le C# copie la référence de l'objet `Voiture` qui est contenue dans la variable `voitureNicolas` dans la variable `voitureJeremie`. Ce sont donc deux variables différentes qui possèdent toutes les deux une référence vers l'objet `Voiture`, qui est la voiture de Nicolas. C'est-à-dire que les deux variables référencent le même objet. Ainsi, la modification des propriétés de l'un affectera forcément l'autre.

Inattendu au premier abord, mais finalement, c'est très logique. Comprendre cette différence entre les types valeur et les types référence est important, nous verrons dans les chapitres suivants quels sont les autres impacts de cette différence. À noter également qu'il est impossible de dériver d'un type intégré alors que c'est possible, et facile, de dériver d'une classe.

D'ailleurs, si nous parlions un peu d'héritage ?

Héritage

Nous avons vu pour l'instant la théorie de l'héritage. Que les objets chiens héritaient des comportements des objets `Animaux`, que les labradors héritaient des comportements des chiens, etc.

Passons maintenant à la pratique et créons une classe `Animal` et une classe `Chien` qui en hérite. Nous allons créer des classes relativement courtes et nous nous limiterons dans le nombre d'actions ou de propriétés de celles-ci. Par exemple, nous pourrions imaginer que la classe `Animal` possède une propriété `NombreDePattes` qui est un entier et une méthode `Respirer` qui affiche le détail de l'action. Ce qui donne :

```
public class Animal
{
    public int NombreDePattes { get; set; }

    public void Respirer()
    {
        Console.WriteLine("Je respire");
    }
}
```

La classe `Chien` dérive de la classe `Animal` et peut donc hériter de certains de ses comportements. En l'occurrence, la classe `Chien` héritera de tout ce qui est public ou protégé, identifiés comme vous le savez désormais par les mots-clés `public` et `protected`. Le chien sait également faire quelque chose qui lui est propre, à savoir aboyer. Il possédera donc une méthode supplémentaire. Ce qui donne :

```
public class Chien : Animal
{
    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}
```

On représente la notion d'héritage en ajoutant après la classe le caractère « : » suivi de la classe mère. Ici, nous avons défini une classe publique `Chien` qui hérite de la classe `Animal`.

Nous pouvons dès à présent créer des objets `Animal` et des objets `Chien`, par exemple :

```
Animal animal = new Animal { NombreDePattes = 4 };
animal.Respirer();
Console.WriteLine();

Chien chien = new Chien { NombreDePattes = 4 };
chien.Respirer();
chien.Aboyer();
```

Si nous exécutons ce code, nous nous rendons bien compte que l'objet `Chien`, bien que n'ayant pas défini la propriété `NombreDePattes` ou la méthode `Respirer()` dans le corps de sa classe, est capable d'avoir des pattes et de faire l'action respirer :

```
Je respire

Je respire
Wouaf !
```

Il a hérité ces comportements de l'objet `Animal`, en tout cas, ceux qui sont publics. Rajoutons deux variables membres de la classe `Animal` :

```
public class Animal
{
    private bool estVivant;
    public int age;

    public int NombreDePattes { get; set; }

    public void Respirer()
    {
        Console.WriteLine("Je respire");
    }
}
```

L'entier `age` est public alors que le booléen `estVivant` est privé. Si nous tentons de les utiliser depuis la classe fille `Chien`, comme ci-dessous :

```
public class Chien : Animal
{
    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }

    public void Vieillir()
    {
        age++;
    }

    public void Naissance()
    {
        age = 0;
        estVivant = true;   /* Erreur   > '
            MaPremiereApplication.Animal.estVivant' est
            inaccessible en raison de son niveau de protection
            */
    }
}
```

nous voyons qu'il est tout à fait possible d'utiliser la variable `age` depuis la méthode `Vieillir()` alors que l'utilisation du booléen `estVivant` provoque une erreur de compilation. Vous avez bien compris que celui-ci était inaccessible car il est défini comme membre privé. Pour l'utiliser, on pourra le rendre public par exemple.

Il existe par contre un autre mot-clé qui permet de rendre des variables, propriétés ou méthodes inaccessibles depuis un autre objet tout en les rendant accessibles depuis des classes filles. Il s'agit du mot-clé `protected`. Si nous l'utilisons à la place de `private` pour définir la visibilité du booléen `estVivant`, nous pourrons nous rendre compte que la classe `Chien` peut désormais compiler :

```
public class Animal
{
    protected bool estVivant;
    [... Extrait de code supprimé ...]
}

public class Chien : Animal
{
    [... Extrait de code supprimé ...]

    public void Naissance()
    {
        age = 0;
        estVivant = true;   // compilation OK
    }
}
```

Par contre, cette variable est toujours inaccessible depuis d'autres classes, comme l'est également une variable privée. Dans notre classe `Program`, l'instruction suivante :

```
chien.estVivant = true;
```

provoquera l'erreur de compilation que désormais nous connaissons bien :

```
'MaPremiereApplication.Animal.estVivant' est inaccessible en
   raison de son niveau de protection
```

Le mot-clé `protected` prend tout son intérêt dès que nous avons à faire avec l'héritage. Nous verrons un peu plus loin d'autres exemples de ce mot-clé.

Nous avons dit dans l'introduction qu'un objet B qui dérive de l'objet A est « une sorte » d'objet A. Dans notre exemple du dessus, le `Chien` est une sorte d'`Animal`. Cela veut dire que nous pouvons utiliser un chien en tant qu'animal. Par exemple, le code suivant :

```
Animal animal = new Chien { NombreDePattes = 4 };
```

est tout à fait correct. Nous disons que notre variable `animal`, de type `Animal` est une instance de `Chien`.

Avec cette façon d'écrire, nous avons réellement instancié un objet `Chien` mais celui-ci sera traité en tant qu'`Animal`. Cela veut dire qu'il sera capable de `Respirer()` et d'avoir des pattes. Par contre, même si en vrai, notre objet est capable d'aboyer, le fait qu'il soit manipulé en tant qu'`Animal` nous empêche de pouvoir le faire `Aboyer`. Cela veut dire que le code suivant :

```
Animal animal = new Chien { NombreDePattes = 4 };
animal.Respirer();
animal.Aboyer(); // erreur de compilation
```

provoquera une erreur de compilation pour indiquer que la classe `Animal` ne contient aucune définition pour la méthode `Aboyer()`. Ce qui est normal, car un animal ne sait pas forcément aboyer. . .



Quel est l'intérêt alors d'utiliser le chien en tant qu'animal ?

Bonne question.

Pour y répondre, nous allons enrichir notre classe `Animal`, garder notre classe `Chien` et créer une classe `Chat` qui hérite également d'`Animal`. Ce pourrait être :

```
public class Animal
{
    protected string prenom;

    public void Respirer()
    {
        Console.WriteLine("Je suis " + prenom + " et je respire
            ");
    }
}

public class Chien : Animal
{
    public Chien(string prenomDuChien)
    {
        prenom = prenomDuChien;
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class Chat : Animal
{
    public Chat(string prenomDuChat)
    {
        prenom = prenomDuChat;
    }

    public void Miauler()
    {
        Console.WriteLine("Miaou");
    }
}
```

Nous forçons les chiens et les chats à avoir un nom, hérité de la classe `Animal`, grâce au constructeur afin de pouvoir les identifier facilement. Le chat garde le même principe que le chien, sauf que nous avons une méthode `Miauler()` à la place de la méthode `Aboyer()`... Ce qui est, somme toute, logique ! L'idée est de pouvoir utiliser nos chiens et nos chats ensemble comme des animaux, par exemple en utilisant une liste.

Pour illustrer ce fonctionnement, donnons vie à quelques chiens et à quelques chats grâce à nos pouvoirs de développeur et mettons-les dans une liste :

```
List<Animal> animaux = new List<Animal>();
Animal milou = new Chien("Milou");
Animal dingo = new Chien("Dingo");
Animal idefix = new Chien("Idéfix");
Animal tom = new Chat("Tom");
Animal felix = new Chat("Félix");

animaux.Add(milou);
animaux.Add(dingo);
animaux.Add(idefix);
animaux.Add(tom);
animaux.Add(felix);
```

Nous avons dans un premier temps instancié une liste d'animaux à laquelle nous avons rajouté 3 chiens et 2 chats, chacun étant considéré comme un animal puisqu'ils sont tous des sortes d'animaux, grâce à l'héritage. Maintenant, nous n'avons plus que des animaux dans la liste. Il sera donc possible de les faire tous respirer en une simple boucle :

```
foreach (Animal animal in animaux)
{
    animal.Respirer();
}
```

Ce qui donne :

```
Je suis Milou et je respire
Je suis Dingo et je respire
Je suis Idéfix et je respire
Je suis Tom et je respire
Je suis Félix et je respire
```

Et voilà, c'est super simple !

Imaginez le bonheur de Noé sur son arche quand il a compris que grâce à la POO, il pouvait faire respirer tous les animaux en une seule boucle ! Quel travail économisé. Peu importe ce qu'il y a dans la liste, des chiens, des chats, des hamsters, nous savons que ce sont tous des animaux et qu'ils savent tous respirer.

Vous avez sans doute remarqué que nous faisons la même chose dans le constructeur de la classe `Chien` et dans celui de la classe `Chat`. Deux fois la même chose... Ce n'est pas terrible. Peut-être y a-t-il un moyen de factoriser tout ça ? Effectivement, il est possible également d'écrire nos classes de cette façon :

```
public class Animal
{
    protected string prenom;

    public Animal(string prenomAnimal)
```

```
    {
        prenom = prenomAnimal;
    }

    public void Respirer()
    {
        Console.WriteLine("Je suis " + prenom + " et je respire
            ");
    }
}

public class Chien : Animal
{
    public Chien(string prenomDuChien) : base(prenomDuChien)
    {
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class Chat : Animal
{
    public Chat(string prenomDuChat) : base(prenomDuChat)
    {
    }

    public void Miauler()
    {
        Console.WriteLine("Miaou");
    }
}
```

Qu'est-ce qui change ?

Eh bien la classe `Animal` possède un constructeur qui prend en paramètre un prénom et qui le stocke dans sa variable privée. C'est elle qui fait le travail d'initialisation. Il devient alors possible pour les constructeurs des classes filles d'appeler le constructeur de la classe mère afin de faire l'affectation du prénom. Pour cela, on utilise les deux points suivis du mot-clé `base` qui signifie « appelle-moi le constructeur de la classe du dessus » auquel nous passons la variable en paramètre. Avec cette écriture un peu barbare, il devient possible de factoriser des initialisations qui ont un sens pour toutes les classes filles. Dans notre cas, je veux que tous les objets qui dérivent d'`Animal` puissent facilement définir un prénom.

Il faut aussi savoir que si nous appelons le constructeur par défaut d'une classe qui n'appelle pas explicitement un constructeur spécialisé d'une classe mère, alors celui-ci appellera automatiquement le constructeur par défaut de la classe dont il hérite.

Modifions à nouveau nos classes pour avoir :

```
public class Animal
{
    protected string prenom;

    public Animal()
    {
        prenom = "Marcel";
    }

    public void Respirer()
    {
        Console.WriteLine("Je suis " + prenom + " et je respire
            ");
    }
}

public class Chien : Animal
{
    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class Chat : Animal
{
    public Chat(string prenomDuChat)
    {
        prenom = prenomDuChat;
    }

    public void Miauler()
    {
        Console.WriteLine("Miaou");
    }
}
```

Ici, la classe `Animal` met un prénom par défaut dans son constructeur. Le chien n'a pas de constructeur et le chat en a un qui accepte un paramètre.

Il est donc possible de créer un `Chien` sans qu'il ait de prénom mais il est obligatoire d'en définir un pour le chat. Sauf que lorsque nous instancierons notre objet `chien`, il appellera automatiquement le constructeur de la classe mère et tous nos chiens s'appelleront Marcel :

```
static void Main(string[] args)
{
    List<Animal> animaux = new List<Animal>();
    Animal chien = new Chien();
    Animal tom = new Chat("Tom");
```

```
    Animal felix = new Chat("Félix");

    animaux.Add(chien);
    animaux.Add(tom);
    animaux.Add(felix);

    foreach (Animal animal in animaux)
    {
        animal.Respirer();
    }
}
```

Ce qui affichera :

```
Je suis Marcel et je respire
Je suis Tom et je respire
Je suis Félix et je respire
```

Il est également possible d'appeler un constructeur à partir d'un autre constructeur. Prenons l'exemple suivant :

```
public class Voiture
{
    private int vitesse;

    public Voiture(int vitesseVoiture)
    {
        vitesse = vitesseVoiture;
    }
}
```

Si nous souhaitons rajouter un constructeur par défaut qui initialise la vitesse à 10 par exemple, nous pourrons faire :

```
public class Voiture
{
    private int vitesse;

    public Voiture()
    {
        vitesse = 10;
    }

    public Voiture(int vitesseVoiture)
    {
        vitesse = vitesseVoiture;
    }
}
```

Ou encore :

```
public class Voiture
{
    private int vitesse;

    public Voiture() : this(10)
    {
    }

    public Voiture(int vitesseVoiture)
    {
        vitesse = vitesseVoiture;
    }
}
```

Ici, l'utilisation du mot-clé `this`, suivi d'un entier permet d'appeler le constructeur qui possède un paramètre entier au début du constructeur par défaut. Inversement, nous pouvons appeler le constructeur par défaut d'une classe depuis un constructeur possédant des paramètres afin de pouvoir bénéficier des initialisations de celui-ci :

```
public class Voiture
{
    private int vitesse;
    private string couleur;

    public Voiture()
    {
        vitesse = 10;
    }

    public Voiture(string couleurVoiture) : this()
    {
        couleur = couleurVoiture;
    }
}
```

Puisque nous parlons d'héritage, il faut savoir que tous les objets que nous créons ou qui sont disponibles dans le framework .NET héritent d'un objet de base. On parle en général d'un « super-objet ». L'intérêt de dériver d'un tel objet est de permettre à tous les objets d'avoir certains comportements en commun, mais également de pouvoir éventuellement tous les traiter en tant qu'objet. Notre super-objet est représenté par la classe `Object` qui définit plusieurs méthodes. Vous les avez déjà vues si vous avez regardé dans la complétion automatique après avoir créé un objet. Prenons une classe toute vide, par exemple :

```
public class ObjetVide
{
}
```

Si nous instancions cet objet et que nous souhaitons l'utiliser, nous verrons que la complétion automatique nous propose des méthodes que nous n'avons jamais créées (voir figure 22.1).

FIGURE 22.1 – La complétion automatique montre des méthodes de la classe de base Object

Nous voyons plusieurs méthodes, comme Equals ou GetHashCode ou GetType ou encore ToString. Comme vous l'avez compris, ce sont des méthodes qui sont définies dans la classe Object. La méthode ToString par exemple permet d'obtenir une représentation de l'objet sous la forme d'une chaîne de caractères. C'est une méthode qui va souvent nous servir, nous y reviendrons un peu plus tard. Ce super-objet est du type Object, mais on utilise généralement son alias object.

Ainsi, il est possible d'utiliser tous nos objets comme des object et ainsi utiliser les méthodes qui sont définies sur la classe Object. Ce qui nous permet de faire :

```
static void Main(string[] args)
{
    ObjetVide monObjetVide = new ObjetVide();
    Chien chien = new Chien();
    int age = 30;
    string prenom = "Nicolas";

    AfficherRepresentation(monObjetVide);
    AfficherRepresentation(chien);
    AfficherRepresentation(age);
    AfficherRepresentation(prenom);
}

private static void AfficherRepresentation(object monObjetVide)
{
    Console.WriteLine(monObjetVide.ToString());
}
```

Ce qui affiche :

```
MaPremiereApplication.ObjetVide
MaPremiereApplication.Chien
30
Nicolas
```

Comme indiqué, la méthode `ToString()` permet d'afficher la représentation par défaut d'un objet. Vous aurez remarqué qu'il y a une différence suivant ce que nous passons. En effet, la représentation par défaut des types référence correspond au nom du type, à savoir son espace de nom suivi du nom de sa classe. Pour ce qui est des types valeur, la représentation contient en général la valeur du type, à l'exception des structures que nous n'avons pas encore vues et que nous aborderons un peu plus loin.

L'intérêt dans cet exemple de code est de voir que nous pouvons tout manipuler comme un `object`. D'une manière générale, vous aurez peu l'occasion de traiter vos objets en tant qu'`object` car il est vraiment plus intéressant de profiter pleinement du type, l'`object` étant peu utilisable.



Notez que l'héritage de `object` est automatique. Nul besoin d'utiliser la syntaxe d'héritage que nous avons déjà vue.

J'en profite maintenant que vous connaissez la méthode `ToString()` pour parler d'un point qui a peut-être titillé vos cerveaux. Dans la première partie, nous avions fait quelque chose du genre :

```
int vitesse = 20;
string chaine = "La vitesse est " + vitesse + " km/h";
```

La variable `vitesse` est un entier. La chaîne `La vitesse est` est une chaîne de caractères. Nous essayons d'ajouter un entier à une chaîne alors que j'ai dit qu'ils n'étaient pas compatibles entre eux ! Et pourtant cela fonctionne. Effectivement, c'est bizarre ! Nous concaténons une chaîne à un entier avec l'opérateur + et nous concaténons encore une chaîne.

Et si je fais l'inverse :

```
int vitesse = 20 + "40";
```

cela provoque une erreur de compilation. C'est logique, on ne peut pas ajouter un entier et une chaîne de caractères. Alors pourquoi cela fonctionne dans l'autre sens ? Ce qui se passe en fait dans l'instruction :

```
string chaine = "La vitesse est " + vitesse + " km/h";
```

c'est que le compilateur se rend compte que nous concaténons une chaîne avec un autre objet, peu importe que ce soit un entier ou un objet complexe. Alors, pour que ça fonctionne, il demande une représentation de l'objet sous la forme d'une chaîne de caractères. Nous avons vu que ceci se faisait en appelant la méthode `ToString()` qui est héritée de l'objet racine `Object`.

L'instruction est donc équivalente à :

```
string chaine = "La vitesse est " + vitesse.ToString() + " km/h
    ";
```

Dans le cas d'un type valeur comme un entier, la méthode `ToString()` renvoie la représentation interne de la valeur, à savoir « 20 ». Dans le cas d'un objet complexe, elle aurait renvoyé le nom du type de l'objet.

Avant de terminer, il est important d'indiquer que le C# n'autorise pas l'héritage multiple. Ainsi, si nous possédons une classe `Carnivore` et une classe `EtreVivant`, il ne sera pas possible de faire hériter directement un objet `Homme` de l'objet `Carnivore` et de l'objet `EtreVivant`. Ainsi, le code suivant :

```
public class Carnivore
{
}
public class EtreVivant
{
}

public class Homme : Carnivore, EtreVivant
{
}
```

provoquera l'erreur de compilation suivante :

```
La classe 'MaPremiereApplication.Homme' ne peut pas avoir
    plusieurs classes de base : 'MaPremiereApplication.Carnivore'
     et 'EtreVivant'
```



Il est impossible de dériver de deux objets en même temps.

En revanche, et c'est pertinent, nous pourrons faire un héritage en cascade afin que `Carnivore` dérive de `EtreVivant` et que `Homme` dérive de `Carnivore` :

```
public class Carnivore : EtreVivant
{
}
public class EtreVivant
{
}

public class Homme : Carnivore
{
}
```

Cependant, il n'est pas toujours pertinent d'opérer de la sorte. Notre `Homme` pourrait être à la fois `Carnivore` et `Frugivore`, cependant cela n'a pas de sens qu'un carnivore soit également frugivore, ou l'inverse.



Tu avais pourtant dit que chaque objet dérivait du super-objet `Object`, mais s'il dérive d'une autre classe comme un chien dérive d'un animal, ça fait bien deux classes dont il dérive...

Effectivement, mais dans ce cas-là, ce n'est pas pareil. Comme il est automatique de dériver de `object`, c'est comme si on avait le `chien` qui hérite de `animal` qui hérite lui-même de `object`. Le C# est assez malin pour ça !

Substitution

Nous avons vu juste avant l'utilisation de la méthode `ToString()` qui permet d'obtenir la représentation d'un objet sous forme de chaîne de caractères. En l'occurrence, vous conviendrez avec moi que la représentation de notre classe `Chien` n'est pas particulièrement exploitable. Le nom du type c'est bien, mais ce n'est pas très parlant.

Ça serait pas mal que, quand nous demandons d'afficher un chien, nous obtenions le nom du chien, vous ne trouvez pas ? C'est là qu'intervient la substitution. Nous en avons parlé dans l'introduction à la POO, la substitution permet de redéfinir un comportement dont l'objet a hérité afin qu'il corresponde aux besoins de l'objet fils. Typiquement, ici, la méthode `ToString()` du super-objet ne nous convient pas et dans le cas de notre chien, nous souhaitons la redéfinir, en écrire une nouvelle version.

Pour cet exemple, simplifions notre classe `Chien` afin qu'elle n'ait qu'une propriété pour stocker son prénom :

```
public class Chien
{
    public string Prenom { get; set; }
}
```

Pour redéfinir la méthode `ToString()` nous allons devoir utiliser le mot-clé `override` qui signifie que nous souhaitons substituer la méthode existante afin de remplacer son comportement, ce que nous pourrons écrire en C# avec :

```
public class Chien
{
    public string Prenom { get; set; }

    public override string ToString()
    {
        return "Je suis un chien et je m'appelle " + Prenom;
    }
}
```

Le mot-clé `override` se met avant le type de retour de la méthode, comme on peut le voir ci-dessus. Si nous appelons désormais la méthode `ToString` de notre objet `Chien` :

```
Chien chien = new Chien { Prenom = "Max" };
Console.WriteLine(chien.ToString());
```

notre programme va utiliser la nouvelle version de la méthode `ToString()` :

```
Je suis un chien et je m'appelle Max
```

Et voilà un bon moyen d'utiliser la substitution, la représentation de notre objet est quand même plus parlante ! Adaptons désormais cet exemple à nos classes. Pour montrer comment faire, reprenons notre classe `Chien` qui possède une méthode `Aboyer()` :

```
public class Chien
{
    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}
```

Nous pourrions imaginer de créer une classe `ChienMuet` qui dérive de la classe `Chien` et qui hérite donc de ses comportements. Mais, que penser d'un chien muet qui serait capable d'aboyer ? Cela n'a pas de sens ! Il faut donc redéfinir cette fichue méthode.

Utilisons alors le mot-clé `override` comme nous l'avons vu pour obtenir :

```
public class ChienMuet : Chien
{
    public override void Aboyer()
    {
        Console.WriteLine("...");
    }
}
```

Créons un chien muet puis faisons-le aboyer, cela donne :

```
ChienMuet pauvreChien = new ChienMuet();
pauvreChien.Aboyer();
```

Sauf que nous rencontrons un problème. Si nous tentons de compiler ce code, Visual C# Express nous génère une erreur de compilation :

```
'MaPremiereApplication.Program.ChienMuet.Aboyer()' : ne peut pas
     substituer le membre hérité 'MaPremiereApplication.Program.
    Chien.Aboyer()', car il n'est pas marqué comme virtual,
    abstract ou override.
```

En réalité, pour pouvoir créer une méthode qui remplace une autre, il faut qu'une condition supplémentaire soit vérifiée : **il faut que la méthode à remplacer s'annonce comme candidate à la substitution**. Cela veut dire que l'on ne peut pas substituer n'importe quelle méthode, mais seulement celles qui acceptent de l'être. C'est le cas pour la méthode `ToString` que nous avons vue précédemment. Les concepteurs du framework .NET ont autorisé cette éventualité. Heureusement, sinon, nous serions bien embêtés !

Pour marquer notre méthode `Aboyer` de la classe `Chien` comme candidate éventuelle à la substitution, il faut la préfixer du mot-clé `virtual`. Ainsi, elle annonce à ses futures filles que si elles le souhaitent, elles peuvent redéfinir cette méthode.

Cela se traduit ainsi dans le code :

```
public class Chien
{
    public virtual void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class ChienMuet : Chien
{
    public override void Aboyer()
    {
        Console.WriteLine("...");
    }
}
```

Désormais, l'instanciation de l'objet est possible et nous pourrons avoir notre code :

```
ChienMuet pauvreChien = new ChienMuet();
pauvreChien.Aboyer();
```

Ce code affichera :

```
...
```

Parfait ! Tout est rentré dans l'ordre.

Le message d'erreur, quoique peu explicite, nous mettait quand même sur la bonne voie. Visual C# Express nous disait qu'il fallait que la méthode soit marquée comme `virtual`, ce que nous avons fait. Il proposait également qu'elle soit marquée `abstract`, nous verrons un peu plus loin ce que ça veut dire. Visual C# Express indiquait enfin que la méthode pouvait être marquée `override`. Cela veut dire qu'une classe fille de `ChienMuet` peut également redéfinir la méthode `Aboyer()` afin qu'elle colle à ses besoins. Elle n'est pas marquée `virtual` mais elle est marquée `override`. Par exemple :

```
public class ChienMuetAvecSyntheseVocale : ChienMuet
{
    public override void Aboyer()
    {
        Console.WriteLine("bwarf !");
    }
}
```

Il y a encore un dernier point que nous n'avons pas abordé. Il s'agit de la capacité pour une classe fille de redéfinir une méthode tout en conservant la fonctionnalité de

la méthode de la classe mère. Imaginons notre classe `Animal` qui possède une méthode `Manger()` :

```
public class Animal
{
    public virtual void Manger()
    {
        Console.WriteLine("Mettre les aliments dans la bouche")
            ;
        Console.WriteLine("Mastiquer");
        Console.WriteLine("Avaler");
        Console.WriteLine("...");
    }
}
```

Notre classe `Chien` pourra s'appuyer sur le comportement de la méthode `Manger()` de la classe `Animal` pour créer sa propre action personnelle. Cela se passe en utilisant à nouveau le mot-clé `base` qui représente la classe mère. Nous pourrons par exemple appeler la méthode `Manger()` de la classe mère afin de réutiliser son fonctionnement. Cela donne :

```
public class Chien : Animal
{
    public override void Manger()
    {
        Console.WriteLine("Réclamer à manger au maître");
        base.Manger();
        Console.WriteLine("Remuer la queue");
    }
}
```

Dans cet exemple, je fais quelque chose avant d'appeler la méthode de la classe mère, puis je fais quelque chose d'autre après. Maintenant, si nous faisons manger notre chien :

```
Chien chien = new Chien();
chien.Manger();
```

s'affichera dans la console toute la série des actions définies comme étant « manger » :

```
Réclamer à manger au maître
Mettre les aliments dans la bouche
Mastiquer
Avaler
...
Remuer la queue
```

Nous voyons bien avec cet exemple comment la classe fille peut réutiliser les méthodes de sa classe mère.



À noter qu'on peut également parler de **spécialisation** ou de **redéfinition** à la place de la substitution.

Polymorphisme

Nous avons dit qu'une manifestation du polymorphisme était la capacité pour une classe d'effectuer la même action sur différents types d'intervenants. Il s'agit de la surcharge, appelée aussi polymorphisme *ad hoc*.

Concrètement, cela veut dire qu'il est possible de définir la même méthode avec des paramètres en entrée différents. Si vous vous rappelez bien, c'est quelque chose que nous avons déjà fait sans le savoir. Devinez ... Oui, c'est ça, avec la méthode `Console.WriteLine`.

Nous avons pu afficher des chaînes de caractères, mais aussi des entiers, même des types `double`, et plus récemment des objets. Comment ceci est possible alors que nous avons déjà vu qu'il était impossible de passer des types en paramètres d'une méthode qui ne correspondent pas à sa signature ?!

Ainsi, l'exemple suivant :

```
public class Program
{
    static void Main(string[] args)
    {
        Math math = new Math();
        int a = 5;
        int b = 6;
        int resultat = math.Addition(a, b);

        double c = 1.5;
        double d = 5.0;
        resultat = math.Addition(c, d); /* erreur de compilation */
    }
}

public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }
}
```

provoquera une erreur de compilation lorsque nous allons essayer de passer des variables du type `double` à notre méthode qui prend des entiers en paramètres. Pour que ceci fonctionne, nous allons rendre polymorphe cette méthode en définissant à nouveau cette même méthode mais en lui faisant prendre des paramètres d'entrée différents :

```
public class Program
{
    static void Main(string[] args)
    {
        Math math = new Math();
```

```
        int a = 5;
        int b = 6;
        int resultat = math.Addition(a, b);

        double c = 1.5;
        double d = 5.0;
        double resultatDouble = math.Addition(c, d); /* ça
            compile, youpi */
    }
}

public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }

    public double Addition(double a, double b)
    {
        return a + b;
    }
}
```

Nous avons ainsi écrit deux formes différentes de la même méthode. Une qui accepte des entiers et l'autre qui accepte des `double`. Ce code fonctionne désormais correctement.

Il est bien sûr possible d'écrire cette méthode avec beaucoup de paramètres de types différents, même une classe `Chien`, en imaginant que le fait d'additionner deux chiens correspond au fait d'additionner leurs nombres de pattes :

```
public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }

    public double Addition(double a, double b)
    {
        return a + b;
    }

    public int Addition(Chien c1, Chien c2)
    {
        return c1.NombreDePattes + c2.NombreDePattes;
    }
}
```

Attention, j'ai toujours indiqué qu'il était possible d'ajouter une nouvelle forme à la même méthode en changeant les paramètres d'entrées. Vous ne pourrez pas le faire en

changeant uniquement le paramètre de retour. Ce qui fait que cet exemple ne pourra pas compiler :

```
public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }

    public double Addition(int a, int b)
    {
        return a + b;
    }
}
```

Les deux méthodes acceptent deux entiers en paramètres et renvoient soit un entier, soit un `double`. Le compilateur ne sera pas capable de choisir quelle méthode utiliser lorsque nous essayerons d'appeler cette méthode. Les méthodes doivent se différencier avec les paramètres d'entrées. Lorsque nous avons plusieurs signatures possibles pour la même méthode, vous remarquerez que la complétion automatique nous propose alors plusieurs possibilités (voir figure 22.2).

FIGURE 22.2 – La complétion automatique propose plusieurs signatures de la même méthode

Visual C# indique qu'il a trois méthodes possibles qui s'appellent `Math.Addition`. Pour voir la signature des autres méthodes, il suffit de cliquer sur les flèches, ou d'utiliser les flèches du clavier, comme indiqué à la figure 22.3.

C'est ce qui se passe dans la méthode `Console.WriteLine` (voir figure 22.4).

Nous voyons ici qu'il existe 19 écritures de la méthode `WriteLine`, la cinquième prenant en paramètre un décimal. Notez que pour écrire plusieurs formes de cette méthode, nous pouvons également jouer sur le nombre de paramètres. La méthode :

```
public int Addition(int a, int b, int c)
{
    return a + b + c;
}
```

```
static void Main(string[] args)
{
    Math math = new Math();
    int a = 5;
    int b = 6;
    int resultat = math.Addition(a, b);

    double c = 1.5;
    double d = 5.0;
    double resultatDouble = math.Addition(
}
```

▲ 2 sur 3 ▼ double Math.Addition(**double a**, double b)

Figure 22.3 – La complétion automatique présente plusieurs signatures de la méthode `Addition`

MaPremiereApplication.Program — Main(string[] args)

```
static void Main(string[] args)
{
    Console.WriteLine(
}
```

▲ 5 sur 19 ▼ void Console.WriteLine(**decimal value**)
Écrit dans le flux de sortie standard la représentation textuelle de la valeur System.Decimal spécifiée, suivie du terminateur de la ligne active.
value: *Valeur à écrire.*

Figure 22.4 – La complétion automatique présente plusieurs signatures de la méthode `Console.WriteLine`

sera bien une nouvelle forme de la méthode `Addition`.

Nous avons également vu dans le chapitre sur les constructeurs d'une classe qu'il était possible de cumuler les constructeurs avec des paramètres différents. Il s'agit à nouveau du polymorphisme. Il nous permet de définir différents constructeurs sur nos objets.

La conversion entre les objets avec le casting

Nous avons déjà vu dans la partie précédente qu'il était possible de convertir les types qui se ressemblent entre eux. Cela fonctionne également avec les objets. Plus précisément, cela veut dire que nous pouvons convertir un objet en un autre seulement s'il est une sorte de l'autre objet. Nous avons vu dans les chapitres précédents qu'il s'agissait de la notion d'héritage.

Ainsi, si nous avons défini une classe `Animal` et que nous définissons une classe `Chien` qui hérite de cette classe `Animal` :

```
public class Animal
{
}

public class Chien : Animal
{
}
```

nous pourrons alors convertir le `chien` en `animal` dans la mesure où le chien est une

sorte d'animal :

```
Chien medor = new Chien();
Animal animal = (Animal)medor;
```

Nous utilisons pour ce faire un cast, comme nous l'avons déjà fait pour les types intégrés (`int`, `bool`, etc.). Il suffit de préfixer la variable à convertir du type entre parenthèses dans lequel nous souhaitons le convertir. Ici, nous pouvons convertir facilement notre `Chien` en `Animal`.

Par contre, il est impossible de convertir un chien en voiture, car il n'y a pas de relation d'héritage entre les deux. Ainsi les instructions suivantes :

```
Chien medor = new Chien();
Voiture voiture = (Voiture)medor;
```

provoqueront une erreur de compilation. Nous avons précédemment utilisé l'héritage afin de mettre des chiens et des chats dans une liste d'animaux. Nous avions fait quelque chose du genre :

```
List<Animal> animaux = new List<Animal>();
Animal chien = new Chien();
Animal chat = new Chat();

animaux.Add(chien);
animaux.Add(chat);
```

Il serait plus logique en fait d'écrire les instructions suivantes :

```
List<Animal> animaux = new List<Animal>();
Chien chien = new Chien();
Chat chat = new Chat();

animaux.Add((Animal)chien);
animaux.Add((Animal)chat);
```

Dans ce cas, nous créons un objet `Chien` et un objet `Chat` que nous mettons dans une liste d'objets `Animal` grâce à une conversion utilisant un cast. En fait, ce cast est inutile et nous pouvons simplement écrire :

```
List<Animal> animaux = new List<Animal>();
Chien chien = new Chien();
Chat chat = new Chat();

animaux.Add(chien);
animaux.Add(chat);
```

La conversion est implicite, comme lorsque nous avions utilisé un `object` en paramètres d'une méthode et que nous pouvions lui passer tous les types qui dérivent d'`object`. Nous avions également vu que nous pouvions traiter les chiens et les chats comme des animaux à partir du moment où nous les mettions dans une liste. Avec les objets suivants :

```
public class Animal
{
    public void Respirer()
    {
        Console.WriteLine("Je respire");
    }
}

public class Chien : Animal
{
    public void Aboyer()
    {
        Console.WriteLine("Waouf");
    }
}

public class Chat : Animal
{
    public void Miauler()
    {
        Console.WriteLine("Miaou");
    }
}
```

Nous pouvions utiliser une boucle pour faire respirer tous nos animaux :

```
List<Animal> animaux = new List<Animal>();
Chien chien = new Chien();
Chat chat = new Chat();

animaux.Add(chien);
animaux.Add(chat);

foreach (Animal animal in animaux)
{
    animal.Respirer();
}
```

Mais impossible de faire aboyer le chien, ni miauler le chat. Si vous tentez de remplacer dans la boucle `Animal` par `Chien`, avec :

```
foreach (Chien c in animaux)
{
    c.Aboyer();
}
```

Vous pourrez faire aboyer le premier élément de la liste qui est effectivement un chien, par contre il y aura un plantage au deuxième élément de la liste car il s'agit d'un chat :

```
Waouf
```

```
Exception non gérée : System.InvalidCastException: Impossible d'
    effectuer un cast d'un objet de type 'MaPremiereApplication.
    Chat' en type 'MaPremiereApplication.Chien'.
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    19
```

Lorsque notre programme a tenté de convertir un animal qui est un chat en chien, il nous a fait comprendre qu'il n'appréciait que moyennement. Les chiens n'aiment pas trop les chats d'une manière générale, alors en plus, un chat qui essaie de se faire passer pour un chien : c'est une déclaration de guerre ! Voilà pourquoi notre programme a levé une exception. Il lui était impossible de convertir un `Chat` en `Chien`.

Il est cependant possible de tester si une variable correspond à un objet grâce au mot-clé `is`. Ce qui nous permettra de faire la conversion adéquate et de nous éviter une erreur à l'exécution :

```
foreach (Animal animal in animaux)
{
    if (animal is Chien)
    {
        Chien c = (Chien)animal;
        c.Aboyer();
    }
    if (animal is Chat)
    {
        Chat c = (Chat)animal;
        c.Miauler();
    }
}
```

Nous testons avec le mot-clé `is` si l'animal est une instance d'un chien ou d'un chat. Le code du dessus nous permettra d'utiliser dans la boucle l'animal courant comme un chien ou un chat en fonction de ce qu'il est vraiment, grâce au test :

```
Waouf
Miaou
```

Le fait de tester ce qu'est vraiment l'animal avant de le convertir est une sécurité indispensable pour éviter ce genre d'erreur. C'est l'inconvénient du cast explicite. Il convient très bien si nous sommes certains du type dans lequel nous souhaitons en convertir un autre. Par contre, si la conversion n'est pas possible, alors nous aurons une erreur. Lorsque nous ne sommes pas certains du résultat du cast, mieux vaut tester si l'instance d'un objet correspond bien à l'objet lui-même.

Cela peut se faire comme nous l'avons vu avec le mot-clé `is`, mais également avec un autre cast qui s'appelle le cast dynamique. Il se fait en employant le mot-clé `as`. Ce cast dynamique vérifie que l'objet est bien convertible. Si c'est le cas, alors il fait un

cast explicite pour renvoyer le résultat de la conversion, sinon, il renvoie une référence nulle. Le code du dessus peut donc s'écrire :

```
foreach (Animal animal in animaux)
{
    Chien c1 = animal as Chien;
    if (c1 != null)
    {
        c1.Aboyer();
    }
    Chat c2 = animal as Chat;
    if (c2 != null)
    {
        c2.Miauler();
    }
}
```

On utilise le mot-clé `as` en le faisant précéder de la valeur à tenter de convertir et en le faisant suivre du type dans lequel nous souhaitons la convertir.

Fonctionnellement, nous faisons la même chose dans les deux codes. Vous pouvez choisir l'écriture que vous préférez, mais sachez que c'est ce dernier qui est en général utilisé, car il est préconisé par Microsoft. De plus, il est un tout petit peu plus performant.

Un petit détail encore. Il est possible de convertir un type valeur, comme un `int` ou un `string` en type référence en utilisant ce qu'on appelle le **boxing**. Rien à voir avec le fait de taper sur les types valeur ! Comme nous l'avons vu, les types valeur et les types référence sont gérés différemment par .NET. Aussi, si nous convertissons un type valeur en type référence, .NET fait une opération spéciale automatiquement. Ainsi le code suivant :

```
int i = 5;
object o = i; // boxing
```

effectue un boxing automatique de l'entier en type référence. C'est ce boxing automatique qui nous permet de manipuler les types valeur comme des `object`. C'est aussi ce qui nous a permis plus haut de passer un entier en paramètre à une méthode qui acceptait un `object`. En interne, ce qui se passe c'est que `object` se voit attribuer une référence vers une copie de la valeur de i. Ainsi, modifier `o` ne modifiera pas `i`. Ce code :

```
int i = 5;
object o = i; // boxing
o = 6;
Console.WriteLine(i);
Console.WriteLine(o);
```

affiche 5 puis 6. Le contraire est également possible, ce qu'on appelle l'**unboxing**. Seulement, celui-ci a besoin d'un cast explicite afin de pouvoir compiler. C'est-à-dire :

```
int i = 5;
object o = i; // boxing
int j = (int)o; // unboxing
```

ici nous reconvertissons la référence vers la valeur de o en entier et nous effectuons à nouveau une copie de cette valeur pour la mettre dans j. Ainsi le code suivant :

```
int i = 5;
object o = i; // boxing
o = 6;
int j = (int)o; // unboxing
j = 7;

Console.WriteLine(i);
Console.WriteLine(o);
Console.WriteLine(j);
```

affichera en toute logique 5 puis 6 puis 7.



À noter que ces opérations sont consommatrices de temps, elles sont donc à faire le moins possible.

En résumé

- Les objets peuvent être des **types valeur** ou des **types référence**. Les variables de type valeur possèdent la valeur de l'objet, comme un entier. Les variables de type référence possèdent une référence vers l'objet en mémoire.
- Tous les objets dérivent de la classe de base `Object`.
- On peut substituer une méthode grâce au mot-clé `override` si elle s'est déclarée candidate à la substitution grâce au mot-clé `virtual`.
- La surcharge est le polymorphisme permettant de faire varier les types des paramètres d'une même méthode ou leurs nombres.
- Il est possible grâce au cast de convertir un type en un autre type, s'ils ont une relation d'héritage.

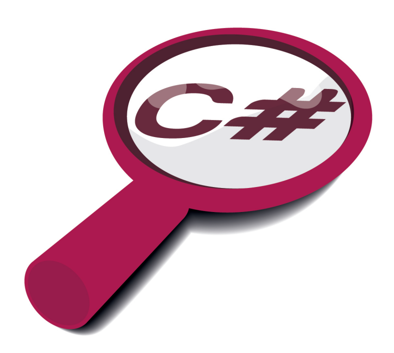
Chapitre 23

Notions avancées de POO en C#

Difficulté : 

Dans ce chapitre, nous allons continuer à découvrir comment nous pouvons faire de l'orienté objet avec le C#. Nous allons pousser un peu plus loin en découvrant les interfaces et en manipulant les classes statiques et abstraites.

À la fin de ce chapitre, vous serez capables de faire des objets encore plus évolués et vous devriez être capables de créer un vrai petit programme orienté objet. . . !



Comparer des objets

Nous avons vu dans la première partie qu'il était possible de comparer facilement des types valeur grâce aux opérateurs de comparaison. En effet, vu que des variables de ces types possèdent directement la valeur que nous lui affectons, on peut facilement comparer un entier avec la valeur 5, ou un entier avec un autre entier. Par contre, cela ne fonctionne pas avec les objets. En effet, nous avons vu que les variables qui représentent des instances d'objet contiennent en fait une référence vers l'instance.

Cela n'a pas vraiment de sens de comparer des références. De plus, en imaginant que je veuille vraiment comparer deux voitures, sur quels critères puis-je déterminer qu'elles sont égales ? La couleur ? La vitesse ?

Sans rien faire, la comparaison en utilisant par exemple l'opérateur d'égalité « == » permet simplement de vérifier si les références pointent vers le même objet.

Pour les exemples de ce chapitre, nous nous baserons sur la classe `Voiture` suivante :

```
public class Voiture
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }
}
```

Ainsi, si nous écrivons :

```
Voiture voitureNicolas = new Voiture();
voitureNicolas.Couleur = "Bleue";
Voiture voitureJeremie = voitureNicolas;
voitureJeremie.Couleur = "Verte";
if (voitureJeremie == voitureNicolas)
{
    Console.WriteLine("Les objets référencent la même instance");
}
```

Ce code affichera la chaîne « Les objets référencent la même instance » car effectivement, nous avons affecté la référence de `voitureNicolas` à `voitureJeremie` - ce qui implique également que la modification de la voiture de Jérémie affecte également la voiture de Nicolas, comme nous l'avons déjà vu.

Par contre, le code suivant :

```
Voiture voitureNicolas = new Voiture();
Voiture voitureJeremie = new Voiture();
if (voitureJeremie == voitureNicolas)
{
    Console.WriteLine("Les objets référencent la même instance");
}
```

n'affichera évidemment rien car ce sont deux instances différentes.

S'il s'avère qu'il est vraiment pertinent de comparer deux voitures entre elles, il faut savoir que c'est quand même possible. La première chose à faire est de définir les critères de comparaison. Par exemple, nous n'avons qu'à dire que deux voitures sont identiques quand la couleur, la marque et la vitesse sont égales. Je sais, c'est un peu irréel, mais c'est pour l'exemple.

Ainsi, nous pourrons par exemple vérifier que deux voitures sont égales avec l'instruction suivante :

```
if (voitureNicolas.Couleur == voitureJeremie.Couleur &&
    voitureNicolas.Marque == voitureJeremie.Marque &&
    voitureNicolas.Vitesse == voitureJeremie.Vitesse)
{
    Console.WriteLine("Les deux voitures sont identiques");
}
```

La comparaison d'égalité entre deux objets, c'est en fait le rôle de la méthode `Equals()` dont chaque objet hérite de la classe mère `Object`. À part pour les types valeur, le comportement par défaut de la méthode `Equals()` est de comparer les références des objets. Seulement, il est possible de définir un comportement plus approprié pour notre classe `Voiture`, grâce à la fameuse spécialisation. Pour plus d'informations sur la méthode `Equals()`, je vous renvoie au code web suivant :

▷ Méthode `Equals`
Code web : 657987

Comme on l'a déjà vu, on utilise le mot-clé `override`. Ceci est possible dans la mesure où la classe `Object` a défini la méthode `Equals` comme virtuelle, avec le mot-clé `virtual`. Ce qui donne :

```
public class Voiture
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }

    public override bool Equals(object obj)
    {
        Voiture v = obj as Voiture;
        if (v == null)
            return false;
        return Vitesse == v.Vitesse && Couleur == v.Couleur &&
            Marque == v.Marque;
    }
}
```

Remarquons que la méthode `Equals` prend en paramètre un `object`. La première chose à faire est donc de vérifier que nous avons réellement une voiture, grâce au cast dynamique. Ensuite, il ne reste qu'à comparer les propriétés de l'instance courante et de l'objet passé en paramètre.

Pour faire une comparaison entre deux voitures, nous pourrons utiliser le code suivant :

```
Voiture voitureNicolas = new Voiture { Vitesse = 10, Marque = "Peugeot", Couleur = "Grise"};
Voiture voitureJeremie = new Voiture { Vitesse = 10, Marque = "Peugeot", Couleur = "Grise" };
if (voitureNicolas.Equals(voitureJeremie))
{
    Console.WriteLine("Les objets ont les mêmes valeurs dans leurs propriétés");
}
```

Nos deux voitures sont identiques car leurs marques, leurs couleurs et leurs vitesses sont identiques :

```
Les objets ont les mêmes valeurs dans leurs propriétés
```

C'est facile de comparer ! Sauf que vous aurez peut-être remarqué que la compilation de ce code provoque un avertissement. Il ne s'agit pas d'une erreur, mais Visual C# Express nous informe qu'il faut faire attention (voir la figure 23.1).

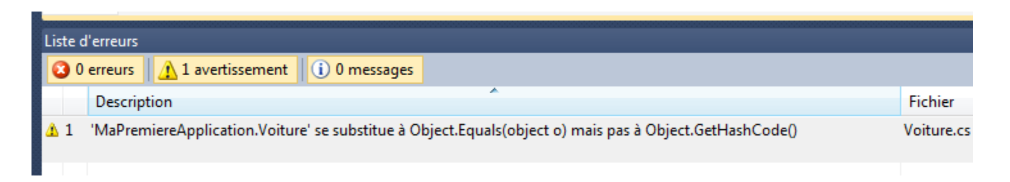


FIGURE 23.1 – La compilation fait apparaître un avertissement

Il nous dit que nous avons substitué la méthode `Equals()` sans avoir redéfini la méthode `GetHashCode()`. Nous n'avons pas besoin ici de savoir à quoi sert vraiment la méthode `GetHashCode()`, mais si l'envie vous prend de vous instruire à ce sujet, vous pouvez lire la documentation officielle disponible via le code web suivant :

▷ Méthode `GetHashCode()`
Code web : 785089

Toujours est-il que nous devons rajouter une spécialisation de la méthode `GetHashCode()`, dont le but est de renvoyer un identifiant plus ou moins unique représentant l'objet, ce qui donnera :

```
public class Voiture
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }

    public override bool Equals(object obj)
    {
        Voiture v = obj as Voiture;
        if (v == null)
            return false;
```

```
            return Vitesse == v.Vitesse && Couleur == v.Couleur &&
                Marque == v.Marque;
        }

        public override int GetHashCode()
        {
            return Couleur.GetHashCode() * Marque.GetHashCode() *
                Vitesse.GetHashCode();
        }
}
```

Nous nous servons du fait que chaque variable de la classe possède déjà un identifiant obtenu avec la méthode `GetHashCode()`. En combinant chaque identifiant de chaque propriété nous pouvons en créer un nouveau. Ici, la classe est complète et prête à être comparée. Elle pourra donc fonctionner correctement avec tous les algorithmes d'égalité du framework .NET, comme les clés de hachage. Notez quand même que devoir substituer ces deux méthodes est une opération relativement rare.

Ce qu'il est important de retenir, c'est ce fameux warning. La conclusion à tirer est que notre façon de comparer, bien que fonctionnelle pour notre voiture, n'est pas parfaite.

Pourquoi ? Parce qu'en ayant substitué la méthode `Equals()`, nous croyons que la comparaison est bonne sauf que le compilateur nous apprend que ce n'est pas le cas. Heureusement qu'il était là, ce compilateur ! Comme c'est une erreur classique, il est capable de la détecter. Mais si c'est autre chose et qu'il ne le détecte pas ?

Tout ça manque d'uniformisation, vous ne trouvez pas ? Il faudrait quelque chose qui nous assure que la classe est correctement comparable. Une espèce de contrat que l'objet s'engagerait à respecter pour être sûr que toutes les comparaisons soient valides.

Un contrat ? Un truc qui finit par « able » ? Ça me rappelle quelque chose ça... mais oui, les interfaces !

Les interfaces

Une fois n'est pas coutume. Plutôt que de commencer par étudier le plus simple, nous allons étudier le plus logique puis nous reviendrons sur le plus simple. C'est-à-dire que nous allons pousser un peu plus loin la comparaison en nous servant des interfaces et nous reviendrons ensuite sur le moyen de créer une interface.

Nous avons donc dit qu'une interface était un contrat que s'engageait à respecter un objet. C'est tout à fait ce dont on a besoin ici. Notre objet doit s'engager à fonctionner pour tous les types de comparaison. Il doit être comparable. Mais comparable ne veut pas forcément dire « égalité », nous devrions être aussi capables d'indiquer si un objet est supérieur à un autre.

Pourquoi ? Imaginons que nous possédions un tableau de voitures et que nous souhaitions le trier comme on a vu dans un chapitre précédent. Pour les entiers c'était une opération plutôt simple ; avec la méthode `Array.Sort()` ils étaient automatiquement triés par ordre croissant. Mais dans notre cas, comment un tableau sera capable de

trier nos voitures ?

Nous voici donc en présence d'un cas concret d'utilisation des interfaces. L'interface `IComparable` permet de définir un contrat de méthodes destinées à la prise en charge de la comparaison entre deux instances d'un objet. Pour la documentation sur cette méthode, je vous renvoie au code web suivant :

▷ IComparable
Code web : 802146

Une fois ces méthodes implémentées, nous serons certains que nos objets seront comparables correctement. Pour cela, nous allons faire en sorte que notre classe « **implémente** » l'interface.

Reprenons notre classe `Voiture` avec uniquement ses propriétés et faisons lui implémenter l'interface `IComparable`. Pour implémenter une interface, on utilisera la même syntaxe que pour hériter d'une classe, c'est-à-dire qu'on utilisera les deux points suivis du type de l'interface. Ce qui donne :

```
public class Voiture : IComparable
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }
}
```



Il est conventionnel de démarrer le nom d'une interface par un I majuscule. C'est le cas de toutes les interfaces du framework .NET.

Si vous tentez de compiler ce code, vous aurez un message d'erreur (voir figure 23.2).

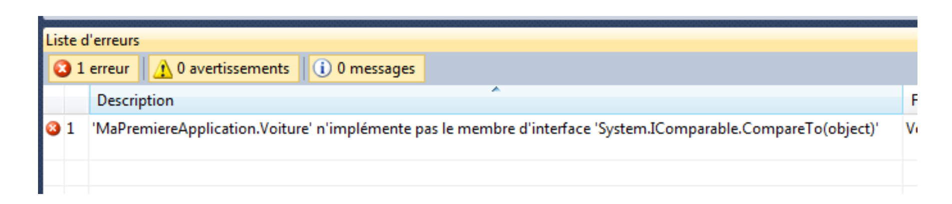


FIGURE 23.2 – Erreur de compilation car l'interface n'est pas implémentée

Le compilateur nous rappelle à l'ordre : nous annonçons que nous souhaitons respecter le contrat de comparaison, sauf que nous n'avons pas la méthode adéquate !

Eh oui, le contrat indique ce que nous nous engageons à faire mais pas la façon de le faire. L'implémentation de la méthode est à notre charge.

Toujours dans l'optique de simplifier la tâche du développeur, Visual C# Express nous aide pour implémenter les méthodes d'un contrat. Faites un clic droit sur l'interface et choisissez dans le menu contextuel `Implémenter l'interface` et `Implémenter l'interface` (voir figure 23.3).

Figure 23.3 – Visual C# Express propose d'implémenter automatiquement l'interface

Visual C# Express nous génère le code suivant :

```
public class Voiture : IComparable
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }

    public int CompareTo(object obj)
    {
        throw new NotImplementedException();
    }
}
```

Il s'agit de la signature de la méthode qui nous manque pour respecter le contrat de l'interface et d'un contenu que nous ne comprenons pas pour l'instant. Nous y reviendrons plus tard ; pour l'instant, vous n'avez qu'à supprimer la ligne :

```
throw new NotImplementedException();
```

Il ne reste plus qu'à écrire le code de la méthode. Pour ce faire, il faut définir un critère de tri. Trier des voitures n'a pas trop de sens, aussi nous dirons que nous souhaitons les trier suivant leurs vitesses.

Pour respecter correctement le contrat, nous devons respecter la règle suivante qui se trouve dans la documentation de la méthode de l'interface :

– si une voiture est inférieure à une autre, alors nous devons renvoyer une valeur inférieure à 0, disons -1 ;
– si elle est égale, alors nous devons renvoyer 0 ;
– si elle est supérieure, nous devons renvoyer une valeur supérieure à 0, disons 1.

Ce qui donne :

```
public int CompareTo(object obj)
{
    Voiture voiture = (Voiture)obj;
    if (this.Vitesse < voiture.Vitesse)
        return -1;
    if (this.Vitesse > voiture.Vitesse)
        return 1;
    return 0;
}
```

La comparaison s'effectue entre l'objet courant et un objet qui lui est passé en paramètre. Pour que ce soit un peu plus clair, j'ai utilisé le mot-clé `this` qui permet de bien identifier l'objet courant et l'objet passé en paramètre. Comme il est facultatif, nous pouvons le supprimer.

Vous aurez également remarqué que j'utilise un cast explicite avant de comparer. Ceci permet de renvoyer une erreur si jamais l'objet à comparer n'est pas du bon type. En effet, que devrais-je renvoyer si jamais l'objet qu'on me passe n'est pas une voiture ? Une erreur ! C'est très bien.

Le code est suffisamment explicite pour que nous comprenions facilement ce que l'on doit faire : comparer les vitesses.

Il est possible de simplifier grandement le code, car pour comparer nos deux voitures, nous effectuons la comparaison sur la valeur d'un entier, ce qui est plutôt trivial. D'autant plus que l'entier, en bon objet comparable, possède également la méthode `CompareTo()`. Ce qui fait qu'il est possible d'écrire notre méthode de comparaison de cette façon :

```
public int CompareTo(object obj)
{
    Voiture voiture = (Voiture)obj;
    return Vitesse.CompareTo(voiture.Vitesse);
}
```

En effet, `Vitesse` étant un type intégré, il implémente déjà correctement la comparaison. C'est d'ailleurs pour ça que le tableau d'entier que nous avons vu précédemment a été capable de se trier facilement.

En ayant implémenté cette interface, nous pouvons désormais trier des tableaux de `Voiture` :

```
Voiture[] voitures = new Voiture[] { new Voiture { Vitesse = 100 }, new Voiture { Vitesse = 40 }, new Voiture { Vitesse = 10 }, new Voiture { Vitesse = 40 }, new Voiture { Vitesse = 50 } };
Array.Sort(voitures);
foreach (Voiture v in voitures)
{
    Console.WriteLine(v.Vitesse);
}
```

Ce qui affichera :

```
10
40
40
50
100
```

Voilà pour le tri, mais si je peux me permettre, je trouve que ce code-là n'est pas très esthétique ! J'y reviendrai un peu plus tard...

Nous avons donc implémenté notre première interface. Finalement, ce n'était pas si compliqué. Voyons à présent comment créer nos propres interfaces. Une interface se définit en C# comme une classe, sauf qu'on utilise le mot-clé `interface` à la place de `class`. En tant que débutant, vous aurez rarement besoin de créer des interfaces. Cependant, il est utile de savoir le faire. Par contre, il sera beaucoup plus fréquent que vos classes implémentent des interfaces existantes du framework .NET, comme nous venons de le faire.

Voyons à présent comment créer une interface et examinons le code suivant :

```
public interface IVolant
{
    int NombrePropulseurs { get; set; }
    void Voler();
}
```



Comme pour les classes, il est recommandé de créer les interfaces dans un fichier à part. Rappelez-vous également de la convention qui fait que les interfaces doivent commencer par un I majuscule.

Nous définissons ici une interface `IVolant` qui possède une propriété de type `int` et une méthode `Voler()` qui ne renvoie rien. Voilà, c'est tout simple !

Nous avons créé une interface. Les objets qui choisiront d'implémenter cette interface seront obligés d'avoir une propriété entière `NombrePropulseurs` et une méthode `Voler()` qui ne renvoie rien. Rappelez-vous que l'interface ne contient que le contrat et aucune implémentation. C'est-à-dire que nous ne verrons jamais de corps de méthode dans une interface ni de variables membres ; uniquement des méthodes et des propriétés. Un contrat.

Notez quand même qu'il ne faut pas définir de visibilité sur les membres d'une interface. Nous serons obligés de définir les visibilités en public sur les objets implémentant l'interface.

Créons désormais deux objets `Avion` et `Oiseau` qui implémentent cette interface, ce qui donne :

```
public class Oiseau : IVolant
{
    public int NombrePropulseurs { get; set; }

    public void Voler()
    {
        Console.WriteLine("Je vole grâce à " +
            NombrePropulseurs + " ailes");
    }
}

public class Avion : IVolant
{
    public int NombrePropulseurs { get; set; }
```

```
    public void Voler()
    {
        Console.WriteLine("Je vole grâce à " +
            NombrePropulseurs + " moteurs");
    }
}
```

Grâce à ce contrat, nous savons maintenant que n'importe lequel de ces objets saura voler.

Il est possible de traiter ces objets comme des objets volants, un peu comme ce que nous avions fait avec les classes mères, en utilisant l'interface comme type pour la variable. Par exemple :

```
IVolant oiseau = new Oiseau { NombrePropulseurs = 2 };
oiseau.Voler();
```

Nous instancions vraiment un objet `Oiseau`, mais nous le manipulons en tant que `IVolant`. Un des intérêts dans ce cas sera de pouvoir manipuler des objets qui partagent un même comportement :

```
Oiseau oiseau = new Oiseau { NombrePropulseurs = 2 };
Avion avion = new Avion { NombrePropulseurs = 4 };

List<IVolant> volants = new List<IVolant> { oiseau, avion };
foreach (IVolant volant in volants)
{
    volant.Voler();
}
```

Ce qui produira :

```
Je vole grâce à 2 ailes
Je vole grâce à 4 moteurs
```

Grâce à l'interface, nous avons pu mettre dans une même liste des objets différents, qui n'héritent pas entre eux mais qui partagent une même interface, c'est-à-dire un même comportement : `IVolant`. Pour accéder à ces objets, nous devrons utiliser leurs interfaces.

Il sera possible quand même de caster nos `IVolant` en `Avion` ou en `Oiseau`, si jamais nous souhaitons rajouter une propriété propre à l'avion. Par exemple je rajoute une propriété `NomDuCommandant` à mon avion mais qui ne fait pas partie de l'interface :

```
public class Avion : IVolant
{
    public int NombrePropulseurs { get; set; }
    public string NomDuCommandant { get; set; }
    public void Voler()
    {
        Console.WriteLine("Je vole grâce à " +
            NombrePropulseurs + " moteurs");
```

```
    }
}
```

Cela veut dire que l'objet `Avion` pourra affecter un nom de commandant mais qu'il ne sera pas possible d'y accéder par l'interface :

```
IVolant avion = new Avion { NombrePropulseurs = 4,
    NomDuCommandant = "Nico" };
Console.WriteLine(avion.NomDuCommandant); // erreur de
    compilation
```

L'erreur de compilation nous indique que `IVolant` ne possède pas de définition pour `NomDuCommandant` ; ce qui est vrai !

Pour accéder au nom du commandant, nous pourrons tenter de caster nos `IVolant` en `Avion`. Si le cast est valide, alors nous pourrons accéder à notre propriété :

```
Oiseau oiseau = new Oiseau { NombrePropulseurs = 2 };
Avion avion = new Avion { NombrePropulseurs = 4,
    NomDuCommandant = "Nico" };

List<IVolant> volants = new List<IVolant> { oiseau, avion };
foreach (IVolant volant in volants)
{
    volant.Voler();
    Avion a = volant as Avion;
    if (a != null)
    {
        Console.WriteLine(a.NomDuCommandant);
    }
}
```

Voilà, c'est tout simple et ça ressemble un peu à ce qu'on a déjà vu.



Lorsque nous avons abordé la boucle `foreach`, j'ai dit qu'elle nous servait à parcourir des éléments « énumérables ». En disant ça, je disais en fait que la boucle `foreach` fonctionne avec tous les types qui implémentent l'interface `IEnumerable`. Maintenant que vous savez ce qu'est une interface, vous comprenez mieux ce à quoi je faisais vraiment référence.

Il faut également noter que les interfaces peuvent hériter entre elles, comme c'est le cas avec les objets. C'est-à-dire que je vais pouvoir déclarer une interface `IVolantMotorise` qui hérite de l'interface `IVolant`.

```
public interface IVolant
{
    int NombrePropulseurs { get; set; }
    void Voler();
}

public interface IVolantMotorise : IVolant
```

```
{
    void DemarrerLeMoteur();
}
```

Ainsi, ma classe `Avion` qui implémentera `IVolantMotorise` devra obligatoirement implémenter les méthodes et propriétés de `IVolant` et la méthode de `IVolantMotorise` :

```
public class Avion : IVolantMotorise
{
    public void DemarrerLeMoteur()
    {
    }

    public int NombrePropulseurs { get; set;}

    public void Voler()
    {
    }
}
```

Enfin, et nous nous arrêterons là pour les interfaces, il est possible pour une classe d'implémenter plusieurs interfaces. Il suffira pour cela de séparer les interfaces par une virgule et d'implémenter bien sûr tout ce qu'il faut derrière. Par exemple :

```
public interface IVolant
{
    void Voler();
}

public interface IRoulant
{
    void Rouler();
}

public class Avion : IVolant, IRoulant
{
    public void Voler()
    {
        Console.WriteLine("Je vole");
    }

    public void Rouler()
    {
        Console.WriteLine("Je Roule");
    }
}
```

Les classes et les méthodes abstraites

Une classe abstraite est une classe particulière qui ne peut pas être instanciée. Concrètement, cela veut dire que nous ne pourrons pas utiliser l'opérateur `new`.

De la même façon, une méthode abstraite est une méthode qui ne contient pas d'implémentation, c'est-à-dire pas de code. Pour être utilisables, les classes abstraites doivent être héritées et les méthodes redéfinies.

En général, les classes abstraites sont utilisées comme classes de base pour d'autres classes. Ces classes fournissent des comportements mais n'ont pas vraiment d'utilité propre. Ainsi, les classes filles qui en héritent pourront bénéficier de leurs comportements et devront éventuellement en remplacer d'autres.

C'est comme une classe incomplète qui ne demande qu'à être complétée.

Si une classe possède une méthode abstraite, alors la classe doit absolument être abstraite. L'inverse n'est pas vrai, une classe abstraite peut posséder des méthodes non abstraites. Vous aurez rarement besoin d'utiliser les classes abstraites en tant que débutant mais elles pourront vous servir pour combiner la puissance des interfaces à l'héritage.

Bon, voilà pour la théorie, passons un peu à la pratique !

Rappelez-vous nos chiens et nos chats qui dérivent d'une classe mère `Animal`. Nous savons qu'un animal est capable de se déplacer. C'est un comportement qu'ont en commun tous les animaux. Il est tout à fait logique de définir une méthode `SeDeplacer()` au niveau de la classe `Animal`. Sauf qu'un chien ne se déplace pas forcément comme un dauphin. L'un a des pattes, l'autre des nageoires. Et même si le chien et le chat semblent avoir un déplacement relativement proche, ils ont chacun des subtilités. Le chat a un mouvement plus gracieux, plus félin.

Bref, on se rend compte que nous sommes obligés de redéfinir la méthode `SeDeplacer()` dans chaque classe fille de la classe `Animal`. Et puis, de toute façon, sommes-nous vraiment capables de dire comment se déplace un `Animal` dans l'absolu ?

La méthode `SeDeplacer` est une candidate parfaite pour une méthode abstraite. Rappelez-vous, la méthode abstraite ne possède pas d'implémentation et chaque classe fille de la classe possédant cette méthode abstraite devra la spécialiser. C'est exactement ce qu'il nous faut.

Pour déclarer une méthode comme étant abstraite, il faut utiliser le mot-clé `abstract` et ne fournir aucune implémentation de la méthode, c'est-à-dire que la déclaration de la méthode doit se terminer par un point-virgule :

```
public class Animal
{
    public abstract void SeDeplacer();
}
```

Si nous tentons de compiler cette classe, nous aurons l'erreur suivante :

```
'MaPremiereApplication.Animal.SeDeplacer()' est abstrait, mais
    est contenu dans la classe non abstraite '
    MaPremiereApplication.Animal'
```

Ah oui, c'est vrai, on a dit qu'une classe qui contient au moins une méthode abstraite était forcément abstraite. C'est le même principe que pour la méthode, il suffit d'utiliser le mot-clé `abstract` devant le mot-clé `class` :

```
public abstract class Animal
{
    public abstract void SeDeplacer();
}
```

Voilà, notre classe peut compiler tranquillement.

Nous avons également dit qu'une classe abstraite pouvait contenir des méthodes concrètes et que c'était d'ailleurs une des grandes forces de ce genre de classes. En effet, il est tout à fait pertinent que des animaux puissent mourir. Et là, ça se passe pour tout le monde de la même façon : le cœur arrête de battre et on ne peut plus rien faire. C'est triste, mais c'est ainsi.

Cela veut dire que nous pouvons créer une méthode `Mourir()` dans notre classe abstraite qui possède une implémentation.

Cela donne :

```
public abstract class Animal
{
    private Coeur coeur;
    public Animal()
    {
        coeur = new Coeur();
    }

    public abstract void SeDeplacer();

    public void Mourir()
    {
        coeur.Stop();
    }
}
```

Avec une classe `Cœur` qui ne fait pas grand-chose :

```
public class Coeur
{
    public void Battre()
    {
        Console.WriteLine("Boom boom");
    }

```

```
    public void Stop()
    {
        Console.WriteLine("Mon coeur s'arrête de battre");
    }
}
```

Comme prévu, il n'est pas possible d'instancier un objet `Animal`. Si nous tentons l'opération :

```
Animal animal = new Animal();
```

nous aurons l'erreur de compilation suivante :

```
Impossible de créer une instance de la classe abstraite ou de l'
   interface 'MaPremiereApplication.Animal'
```

Par contre, il est possible de créer une classe `Chien` qui dérive de la classe `Animal` :

```
public class Chien : Animal
{
}
```

Cette classe ne pourra pas compiler dans cet état car il faut obligatoirement redéfinir la méthode abstraite `SeDeplacer()`. Cela se fait en utilisant le mot-clé `override`, comme on l'a déjà vu.

Vous aurez sûrement remarqué que la méthode abstraite n'utilise pas le mot-clé `virtual` comme cela doit absolument être le cas lors de la substitution d'une méthode dans une classe non-abstraite. Il est ici implicite et ne doit pas être utilisé, sinon nous aurons une erreur de compilation. Et puis cela nous arrange ; un seul mot-clé, c'est largement suffisant !

Nous devons donc spécialiser la méthode `SeDeplacer`, soit en écrivant la méthode à la main, soit en utilisant encore une fois notre ami Visual C# Express.

Il suffit de faire un clic droit sur la classe dont hérite `Chien` (en l'occurrence `Animal`) et de cliquer sur `Implémenter une classe abstraite` (voir la figure 23.4).

FIGURE 23.4 – Visual C# Express propose d'implémenter la classe abstraite

Visual C# Express nous génère donc la signature de la méthode à substituer :

```
public class Chien : Animal
```

```
{
    public override void SeDeplacer()
    {
        throw new NotImplementedException();
    }
}
```

Il ne reste plus qu'à écrire le corps de la méthode `SeDeplacer`, par exemple :

```
public class Chien : Animal
{
    public override void SeDeplacer()
    {
        Console.WriteLine("Waouf ! Je me déplace avec mes 4
            pattes");
    }
}
```

Ainsi, nous pourrons créer un objet `Chien` et le faire se déplacer puis le faire mourir car il hérite des comportements de la classe `Animal`. Paix à son âme.

```
Chien max = new Chien();
max.SeDeplacer();
max.Mourir();
```

Ce code affichera :

```
Waouf ! Je me déplace avec mes 4 pattes
Mon coeur s'arrête de battre
```

Vous pouvez désormais vous rendre compte de la véracité de la phrase que j'ai écrite en introduction :

« [Les classes abstraites] pourront vous servir pour combiner la puissance des interfaces à l'héritage. »

En effet, la classe abstraite peut fournir des implémentations alors que l'interface ne propose qu'un contrat. Cependant, une classe concrète ne peut hériter que d'une seule classe mais peut implémenter plusieurs interfaces.

La classe abstraite est un peu à mi-chemin entre l'héritage et l'interface.

Les classes partielles

Les classes partielles offrent la possibilité de définir une classe en plusieurs fois. En général, ceci est utilisé pour définir une classe sur plusieurs fichiers, si par exemple la classe devient très longue. Il pourra éventuellement être judicieux de découper la classe en plusieurs fichiers pour regrouper des fonctionnalités qui se ressemblent.

On utilise pour cela le mot-clé `partial`.

Par exemple, si nous avons un fichier qui contient la classe `Voiture` suivante :

```
public partial class Voiture
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }
}
```

Nous pourrons compléter sa définition dans un autre fichier pour lui rajouter par exemple des méthodes :

```
public partial class Voiture
{
    public string Rouler()
    {
        return "Je roule à " + Vitesse + " km/h";
    }
}
```

À la compilation, Visual C# Express réunit les deux classes en une seule et l'objet fonctionne comme toute autre classe qui ne serait pas forcément partielle.

À noter qu'il faut impérativement que les deux classes possèdent le mot-clé `partial` pour que cela soit possible, sinon Visual C# Express générera une erreur de compilation :

```
Modificateur partiel manquant sur la déclaration de type '
   MaPremiereApplication.Voiture' ; une autre déclaration
   partielle de ce type existe
```

Vous allez me dire que ce n'est pas super utile comme fonctionnalité. Et je vous dirai que vous avez raison. Sauf dans un cas particulier.

Les classes partielles prennent de l'intérêt quand une partie du code de la classe est générée par Visual C# Express. C'est le cas pour la plupart des plateformes qui servent à développer de vraies applications. Par exemple ASP.NET pour un site internet, WPF pour une application Windows, Silverlight pour un client riche, etc. C'est aussi le cas lorsque nous générons de quoi permettre l'accès à une base de données.

Dans ce cas-là, disons pour simplifier que Visual C# Express va nous générer tout un tas d'instructions pour nous connecter à la base de données ou pour récupérer des données. Toutes ces instructions seront mises dans une classe partielle que nous pourrons enrichir avec nos besoins.

L'intérêt est que si nous générons une nouvelle version de notre classe, seul le fichier généré sera impacté. Si nous avions modifié le fichier pour enrichir la classe avec nos besoins, nous aurions perdu tout notre travail. Vu que, grâce aux classes partielles, ce code est situé dans un autre fichier, il n'est donc pas perdu. Pour notre plus grand bonheur !

À noter que le mot-clé `partial` peut se combiner sans problème avec d'autres mots-clés, comme `abstract` par exemple que nous venons de voir.

Il est fréquent aussi de voir des classes partielles utilisées quand plusieurs développeurs travaillent sur la même classe. Le fait de séparer la classe en deux fichiers permet de travailler sans se marcher dessus.

Nous aurons l'occasion de voir des classes partielles générées plus tard dans l'ouvrage.

Classes statiques et méthodes statiques

Nous avons déjà vu le mot-clé `static` en première partie de ce livre. Il nous a bien encombrés ! Nous nous le sommes trimballé pendant un moment, puis il a disparu.

Il est temps de revenir sur ce mot-clé afin de comprendre exactement de quoi il s'agit, maintenant que nous avons plus de notions et que nous connaissons les objets et les classes.

Jusqu'à présent, nous avons utilisé le mot-clé `static` uniquement sur les méthodes et je l'ai expliqué vaguement en disant qu'il servait à indiquer que la méthode est toujours disponible et prête à être utilisée.

Pas très convaincante mon explication... mais comme vous êtes polis, vous ne m'avez rien dit !

En fait, le mot-clé `static` permet d'indiquer que la méthode d'une classe n'appartient pas à une instance de la classe. Nous avons vu que jusqu'à présent, nous devions instancier nos classes avec le mot-clé `new` pour avoir des objets.

Ici, `static` permet de ne pas instancier l'objet mais d'avoir accès à cette méthode en dehors de tout objet. Nous avons déjà utilisé beaucoup de méthodes statiques, je ne sais pas si vous avez fait attention, mais maintenant que vous connaissez les objets, la méthode suivante ne vous paraît pas bizarre ?

```
Console.WriteLine("Bonjour");
```

Nous utilisons la méthode `WriteLine` de la classe `Console` sans avoir créé d'objet `Console`. Étrange.

Il s'agit, vous l'aurez deviné, d'une méthode statique qui est accessible en dehors de toute instance de `Console`. D'ailleurs, la classe entière est une classe statique. Si nous essayons d'instancier la classe `Console` avec :

```
Console c = new Console();
```

Nous aurons les messages d'erreurs suivants :

```
Impossible de déclarer une variable de type static 'System.
    Console'
Impossible de créer une instance de la classe static 'System.
    Console'
```

Nous avons dit que la méthode spéciale `Main()` est obligatoirement statique. Cela permet au CLR qui va exécuter notre application de ne pas avoir besoin d'instancier

la classe `Program` pour démarrer notre application. Il a juste à appeler la méthode `Program.Main()` afin de démarrer notre programme.

Comme la méthode `Main()` est utilisable en dehors de toute instance de classe, elle ne peut appeler que des méthodes statiques. En effet, comment pourrait-elle appeler des méthodes d'un objet alors qu'elle n'en a même pas conscience.

C'est pour cela que nous avons été obligés de préfixer chacune de nos premières méthodes par le mot-clé `static`.

Revenons à nos objets. Ils peuvent contenir des méthodes statiques ou des variables statiques. Si une classe ne contient que des choses statiques alors elle peut devenir également statique.

Une méthode statique est donc une méthode qui ne travaille pas avec les membres (variables ou autres) non statiques de sa propre classe. Rappelez-vous, un peu plus haut, nous avions créé une classe `Math` qui servait à faire des additions, afin d'illustrer le polymorphisme :

```
public class Math
{
    public int Addition(int a, int b)
    {
        return a + b;
    }
}
```

que nous utilisions de cette façon :

```
Math math = new Math();
int resultat = math.Addition(5, 6);
```

Ici, la méthode `addition` sert à additionner deux entiers. Elle est complètement indépendante de la classe `Math` et donc des instances de l'objet `Math`.

Nous pouvons donc en faire une méthode statique, pour cela il suffira de préfixer du mot-clé `static` son type de retour :

```
public class Math
{
    public static int Addition(int a, int b)
    {
        return a + b;
    }
}
```

Et nous pourrons alors utiliser l'addition sans créer d'instance de la classe `Math`, mais simplement en utilisant le nom de la classe suivi du nom de la méthode statique :

```
int resultat = Math.Addition(5, 6);
```

Exactement comme nous avons fait pour `Console.WriteLine`.

De la même façon, nous pouvons rajouter d'autres méthodes, comme la multiplication :

```
public class Math
{
    public static int Addition(int a, int b)
    {
        return a + b;
    }

    public static long Multiplication(int a, int b)
    {
        return a * b;
    }
}
```

Nous l'appellerons de la même façon :

```
long resultat = Math.Multiplication(5, 6);
```

À noter que la classe `Math` est toujours instanciable mais qu'il n'est pas possible d'appeler les méthodes qui sont statiques depuis un objet `Math`. Le code suivant :

```
Math math = new Math();
long resultat = math.Multiplication(5, 6);
```

provoquera l'erreur de compilation :

```
Le membre 'MaPremiereApplication.Math.Multiplication(int, int)'
   est inaccessible avec une référence d'instance ; qualifiez-le
   avec un nom de type
```



Ok, mais à quoi ça sert de pouvoir instancier un objet `Math` si nous ne pouvons accéder à aucune de ses méthodes, vu qu'elles sont statiques ?

Absolument à rien ! Si une classe ne possède que des membres statiques, alors il est possible de rendre cette classe statique grâce au même mot-clé.

Celle-ci deviendra non-instanciable, comme la classe `Console` :

```
public static class Math
{
    public static int Addition(int a, int b)
    {
        return a + b;
    }
}
```

Ainsi, si nous tentons d'instancier l'objet `Math`, nous aurons une erreur de compilation.

En général, les classes statiques servent à regrouper des méthodes utilitaires qui partagent une même fonctionnalité. Ici, la classe `Math` permettrait de ranger toutes les méthodes du style addition, multiplication, racine carrée, etc.

Ah, on me fait signe que cette classe existe déjà dans le framework .NET et qu'elle s'appelle également `Math`. Elle est rangée dans l'espace de nom `System`. Souvenez-vous, nous l'avons utilisée pour calculer la racine carrée. Cette classe est statique, c'est la version aboutie de la classe que nous avons commencé à écrire.

Par contre, ce n'est pas parce qu'une classe possède des méthodes statiques qu'elle est obligatoirement statique. Il est aussi possible d'avoir des membres statiques dans une classe qui possède des membres non statiques.

Par exemple notre classe `Chien`, qui possède un prénom et qui sait aboyer :

```
public class Chien
{
    private string prenom;

    public Chien(string prenomDuChien)
    {
        prenom = prenomDuChien;
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf ! Je suis " + prenom);
    }
}
```

pourrait posséder une méthode permettant de calculer l'âge d'un chien dans le référentiel des humains.

Comme beaucoup le savent, il suffit de multiplier l'âge du chien par 7 !

```
public class Chien
{
    private string prenom;

    public Chien(string prenomDuChien)
    {
        prenom = prenomDuChien;
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf ! Je suis " + prenom);
    }

    public static int CalculerAge(int ageDuChien)
    {
        return ageDuChien * 7;
    }
}
```

Ici, la méthode `CalculerAge()` est statique car elle ne travaille pas directement avec une instance d'un chien.

Nous pouvons l'appeler ainsi :

```
Chien hina = new Chien("Hina");
hina.Aboyer();
int ageReferentielHomme = Chien.CalculerAge(4);
Console.WriteLine(ageReferentielHomme);
```

Vous aurez ainsi dans la console :

```
Waouf ! Je suis Hina
28
```

Vous me direz qu'il est possible de faire en sorte que la méthode travaille sur une instance d'un objet `Chien`, ce qui serait peut-être plus judicieux ici. Il suffirait de rajouter une propriété `Age` au `Chien` et de transformer la méthode pour qu'elle ne soit plus statique. Ce qui donnerait :

```
public class Chien
{
    private string prenom;

    public int Age { get; set; }

    public Chien(string prenomDuChien)
    {
        prenom = prenomDuChien;
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf ! Je suis " + prenom);
    }

    public int CalculerAge()
    {
        return Age * 7;
    }
}
```

Que l'on pourrait appeler de cette façon :

```
Chien hina = new Chien("Hina") { Age = 5 };
int ageReferentielHomme = hina.CalculerAge();
Console.WriteLine(ageReferentielHomme);
```

Ici, c'est plus une histoire de conception. C'est à vous de décider, mais sachez que c'est possible. Il est également possible d'utiliser le mot-clé `static` avec des propriétés. Imaginons que nous souhaitions avoir un compteur sur le nombre d'instances de la classe `Chien`. Nous pourrions créer une propriété statique de type entier qui s'incrémente à chaque fois que l'on crée un nouveau chien :

```
public class Chien
{
    public static int NombreDeChiens { get; set; }

    private string prenom;

    public Chien(string prenomDuChien)
    {
        prenom = prenomDuChien;
        NombreDeChiens++;
    }

    public void Aboyer()
    {
        Console.WriteLine("Wouaf ! Je suis " + prenom);
    }
}
```

Ici, la propriété `NombreDeChiens` est statique et s'incrémente à chaque passage dans le constructeur.

Ainsi, si nous créons plusieurs chiens :

```
Chien chien1 = new Chien("Max");
Chien chien2 = new Chien("Hina");
Chien chien3 = new Chien("Laika");

Console.WriteLine(Chien.NombreDeChiens);
```

Nous pourrons voir combien de fois nous sommes passés dans le constructeur :

```
3
```

À noter que pour une variable statique, cela se passe de la même façon qu'avec les propriétés statiques.

Les classes internes

Les classes internes [1] sont un mécanisme qui permet d'avoir des classes définies à l'intérieur d'autres classes.

Cela peut être utile si vous souhaitez restreindre l'accès d'une classe uniquement à sa classe mère.

Par exemple :

```
public class Chien
{
    private Coeur coeur = new Coeur();
```

1. Classe interne se dit *nested class* en anglais.

```

    public void Mourir()
    {
        coeur.Stop();
    }

    private class Coeur
    {
        public void Stop()
        {
            Console.WriteLine("The end");
        }
    }
}
```

Ici, la classe `Coeur` ne peut être utilisée que par la classe `Chien` car elle est privée. Nous pourrions également mettre la classe en `protected` afin qu'une classe dérivée de la classe `Chien` puisse également utiliser la classe `Coeur` :

```
public class ChienSamois : Chien
{
    private Coeur autreCoeur = new Coeur();
}
```

Avec ces niveaux de visibilité, une autre classe comme la classe `Chat` ne pourra pas se servir de ce cœur. Si nous mettons la classe `Coeur` en `public` ou `internal`, elle sera utilisable par tout le monde, comme une classe normale. Dans ce cas, pour nos chats, nous pourrions avoir :

```
public class Chat
{
    private Chien.Coeur coeur = new Chien.Coeur();

    public void Mourir()
    {
        coeur.Stop();
    }
}
```

Notez que nous préfixons la classe par le nom de la classe qui contient la classe `Coeur`. Dans ce cas, l'intérêt d'utiliser une classe interne est moindre. Cela permet éventuellement de regrouper les classes de manière sémantique, et encore, c'est plutôt le but des espaces de nom. Voilà pour les classes internes. C'est une fonctionnalité qui est peu utilisée, mais vous la connaissez désormais !

Les types anonymes et le mot-clé `var`

Le mot-clé `var` est un truc de feignant !

Il permet de demander au compilateur de déduire le type d'une variable au moment où nous la déclarons. Par exemple le code suivant :

```
var prenom = "Nicolas";
var age = 30;
```

est équivalent au code suivant :

```
string prenom = "Nicolas";
int age = 30;
```

Le mot-clé `var` sert à indiquer que nous ne voulons pas nous préoccuper de ce qu'est le type et que c'est au compilateur de le trouver.

Cela implique qu'il faut que la variable soit initialisée (et non nulle) au moment où elle est déclarée afin que le compilateur puisse déduire le type de la variable, en l'occurrence, il devine qu'en lui affectant « Nicolas », il s'agit d'une chaîne de caractères.

Je ne recommande pas l'utilisation de ce mot-clé car le fait de ne pas mettre le type de la variable fait perdre de la clarté au code. Ici, c'est facile. On déduit facilement nous aussi que le type de `prenom` est `string` et que le type de `age` est `int`.

Mais c'est aussi parce qu'on est trop fort !

Par contre, si jamais la variable est initialisée grâce à une méthode :

```
var calcul = GetCalcul();
```

il va falloir aller regarder la définition de la méthode afin de savoir le type de retour et connaître ainsi le type de la variable. Des contraintes dont on n'a pas besoin alors qu'il est aussi simple d'indiquer le vrai type et que c'est d'autant plus lisible.



Mais alors, pourquoi en parler ?

Parce que ce mot-clé peut servir dans un cas précis, celui des types anonymes. Lorsqu'il est conjointement utilisé avec l'opérateur `new`, il permet de créer des types anonymes, c'est-à-dire des types dont on n'a pas défini la classe au préalable. Une classe sans nom.

Cette classe est déduite grâce à son initialisation :

```
var unePersonneAnonyme = new { Prenom = "Nico", Age = 30 };
```

Ici nous créons une variable qui contient une propriété `Prenom` et une propriété `Age`. Forcément, il est impossible de donner un type à cette variable vu qu'elle n'a pas de définition. C'est pour cela qu'on utilise le mot-clé `var`.

Ici, on sait juste que la variable `unePersonneAnonyme` possède deux propriétés, un prénom et un âge. En interne, le compilateur va générer un nom de classe pour ce type anonyme, mais il n'a pas de sens pour nous. En l'occurrence, si nous écrivons le code suivant où `GetType()` (hérité de la classe `object`) renvoie le nom du type :

```
var unePersonneAnonyme = new { Prenom = "Nico", Age = 30 };
Console.WriteLine(unePersonneAnonyme.GetType());
```

nous aurons dans la console le type de notre variable :

```
<>f__AnonymousType0`2[System.String,System.Int32]
```

Ce qui est effectivement sans intérêt pour nous !

Jusqu'ici, les types anonymes peuvent sembler ne pas apporter d'intérêt, tant il est simple de définir une classe `Personne` possédant une propriété `Prenom` et une propriété `Age`.

Mais cela permet d'utiliser ces classes comme des classes à usage unique lorsque nous ne souhaitons pas nous encombrer d'un fichier possédant une classe qui va nous servir uniquement à un seul endroit.

Paresse, souci de clarté du code... tout ceci est un peu mêlé dans la création d'un type anonyme. À vous de l'utiliser quand bon vous semble. Vous verrez plus tard que les types anonymes sont souvent utilisés dans les méthodes d'extensions Linq. Nous en reparlerons !

À noter que lorsque nous créons un tableau, par exemple :

```
string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "
    Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

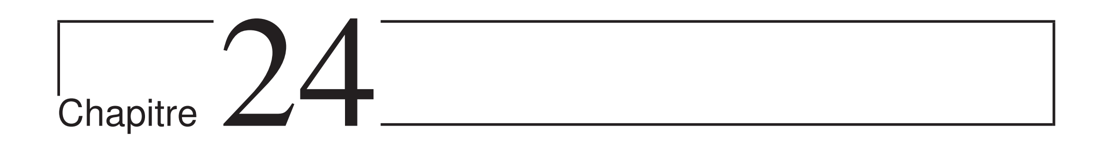
il est également possible de l'écrire ainsi :

```
string[] jours = new[] { "Lundi", "Mardi", "Mercredi", "Jeudi",
     "Vendredi", "Samedi", "Dimanche" };
```

c'est-à-dire sans préciser le type du tableau après le `new`. Le compilateur déduit le type à partir de l'initialisation. Ce n'est pas un type anonyme en soi, mais le principe de déduction est le même.

En résumé

– Une interface est un contrat que s'engage à respecter un objet.
– Il est possible d'implémenter plusieurs interfaces dans une classe.
– Une classe abstraite est une classe qui possède au moins une méthode ou propriété abstraite. Elle ne peut pas être instanciée.
– Une classe concrète dérivant d'une classe abstraite est obligée de substituer les membres abstraits.
– Il est possible de créer des types anonymes grâce à l'opérateur `new` suivi de la description des propriétés de ce type. Les instances sont manipulées grâce au mot-clé `var`.



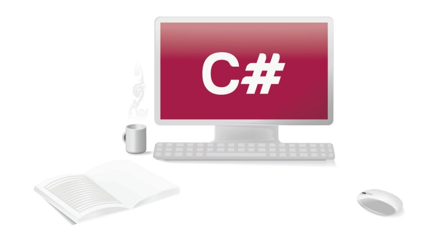
TP : programmation orientée objet

Difficulté : 

Dans ce TP, nous allons essayer de mettre en pratique ce que nous avons appris en programmation orientée objet avec le C#. Il est difficile d'avoir un exercice faisant appel à toutes les notions que nous avons apprises. Aussi, certaines sont laissées de côté.

Le TP est intéressant pour s'entraîner à créer des classes, à manipuler l'héritage et à se confronter à des situations un peu différentes de la théorie.

Le but de ce TP est de créer une mini application de gestion bancaire où nous pourrons gérer des comptes pouvant effectuer des opérations bancaires entre eux. Ne vous attendez pas non plus à refaire les applications de la Banque de France ; on est là pour s'entraîner !



Instructions pour réaliser le TP

Voici l'énoncé de la première partie du TP : dans notre mini application bancaire tous les montants utilisés seront des décimaux. Nous allons devoir gérer des comptes bancaires. Un compte est composé :

- d'un **Solde** (calculé, mais non modifiable) ;
- d'un **Propriétaire** (nom du propriétaire du compte) ;
- d'une **liste d'opérations** internes permettant de garder l'historique du compte, non accessible par les autres objets ;
- d'une méthode permettant de **Crediter()** le compte, prenant une somme en paramètre ;
- d'une méthode permettant de **Crediter()** le compte, prenant une somme et un compte en paramètres, créditant le compte et débitant le compte passé en paramètres ;
- d'une méthode permettant de **Debiter()** le compte, prenant une somme en paramètre ;
- d'une méthode permettant de **Debiter()** le compte, prenant une somme et un compte bancaire en paramètres, débitant le compte et créditant le compte passé en paramètre ;
- d'une méthode qui permet d'afficher le résumé d'un compte.

Un compte courant est **une sorte de compte** et se compose :

- de tout ce qui compose un compte ;
- d'un **découvert autorisé**, non modifiable, défini à l'ouverture du compte.

Le résumé d'un compte courant affiche le solde, le propriétaire, le découvert autorisé ainsi que les opérations sur le compte.

Un compte épargne entreprise est **une sorte de compte** et se compose :

- de tout ce qui compose un compte ;
- d'un **taux d'abondement**, défini à l'ouverture du compte en fonction de l'ancienneté du salarié. Un taux est un `double` compris entre 0 et 1 (5% = 0.05).

Le solde doit tenir compte du taux d'abondement, mais l'abondement n'est pas calculé lors des transactions car l'entreprise verse l'abondement quand elle le souhaite.

Le résumé d'un compte épargne entreprise affiche le solde, le propriétaire, le taux d'abondement ainsi que les opérations sur le compte.

Une opération bancaire est composée :

- d'un montant ;
- d'un type de mouvement, crédit ou débit.

Voilà ! Vous allez maintenant devoir créer une telle application avec les informations suivantes :

- le compte courant de Nicolas a un découvert autorisé de 2000 € ;
- le compte épargne entreprise de Nicolas a un taux de 2% ;
- le compte courant de Jérémie a un découvert autorisé de 500 € ;

- Nicolas touche son salaire de 100 € (pas cher payé) !
- il fait le plein de sa voiture : 50 € ;
- il met de côté sur son compte épargne entreprise la coquette somme de 20 € ;
- il reçoit un cadeau de la banque de 100 €, car il a ouvert son compte épargne pendant la période promotionnelle ;
- il remet ses 20 € sur son compte bancaire, car finalement, il ne se sent pas trop en sécurité !
- Jérémie achète un nouveau PC : 500 € ;
- Jérémie rembourse ses dettes à Nicolas : 200 €.

L'application doit indiquer les soldes de chacun de ces comptes :

```
Solde compte courant de Nicolas : 250
Solde compte épargne de Nicolas : 102,00
Solde compte courant de Jérémie : -700
```

Ensuite, nous afficherons le résumé du compte courant de Nicolas et de son compte épargne entreprise. Ce qui nous donnera :

```
Résumé du compte de Nicolas
*******************************************
Compte courant de Nicolas
        Solde : 250
        Découvert autorisé : 2000

Opérations :
        +100
        -50
        -20
        +20
        +200
*******************************************

Résumé du compte épargne de Nicolas
#####################################
Compte épargne entreprise de Nicolas
        Solde : 102,00
        Taux : 0,02

Opérations :
        +20
        +100
        -20
#####################################
```

Bon, j'ai bien expliqué, ce n'est pas si compliqué, sauf si bien sûr vous n'avez pas lu ou pas compris les chapitres précédents. Dans ce cas, n'hésitez pas à y rejeter un coup d'œil.

Sinon, il suffit de bien décomposer tout en créant les classes les unes après les autres.

Bon courage. . . !

Correction

Ne regardez pas tout de suite la correction ! Prenez votre temps pour faire ce TP, car il est important de savoir bien manipuler les classes. Vous allez faire ça très régulièrement ; ça doit devenir un réflexe, il faut donc pratiquer !

Toujours est-il que voici ma correction. Le plus dur était certainement de modéliser correctement l'application.

Il y a plusieurs solutions bien sûr pour créer ce petit programme, voici celle que je propose.

Tout d'abord, nous devons manipuler des comptes courants et des comptes épargne entreprise, qui sont des sortes de comptes.

On en déduit qu'un compte courant hérite d'un compte. De même, un compte épargne entreprise hérite également d'un compte. D'ailleurs, un compte a-t-il vraiment une raison d'être à part entière ? Nous ne créons jamais de compte « généraliste », seulement des comptes spécialisés. Nous allons donc créer la classe `Compte` en tant que classe abstraite.

Ce qui nous donnera les classes suivantes :

```
public abstract class Compte
{
}

public class CompteCourant : Compte
{
}

public class CompteEpargneEntreprise : Compte
{
}
```

Nous avons également dit qu'un compte était composé d'une liste d'opérations qui possèdent un montant et un type de mouvement. Le type de mouvement pouvant prendre deux valeurs, il paraît logique d'utiliser une énumération :

```
public enum Mouvement
{
    Credit,
    Debit
}
```

La classe d'opération étant :

```
public class Operation
{
    public Mouvement TypeDeMouvement { get; set; }
```

```
    public decimal Montant { get; set; }
}
```

Voilà pour la base.

Ensuite, la classe `Compte` doit posséder un nom de propriétaire et un solde qui peut être redéfini dans la classe `CompteEpargneEntreprise`. Ce solde est calculé à partir d'une liste d'opérations. Le solde est donc une propriété en lecture seule, virtuelle car nécessitant d'être redéfinie :

```
public abstract class Compte
{
    protected List<Operation> listeOperations;
    public string Proprietaire { get; set; }

    public virtual decimal Solde
    {
        get
        {
            decimal total = 0;
            foreach (Operation operation in listeOperations)
            {
                if (operation.TypeDeMouvement == Mouvement.
                    Credit)
                    total += operation.Montant;
                else
                    total -= operation.Montant;
            }
            return total;
        }
    }
}
```

La liste des opérations est une variable membre, déclarée en `protected` car nous allons en avoir besoin dans la classe `CompteCourant` qui en hérite, afin d'afficher un résumé des opérations.

La propriété `Solde` n'est pas très compliquée en soit, il suffit de parcourir la liste des opérations et en fonction du type de mouvement, ajouter ou retrancher le montant. Comme c'est une propriété en lecture seule, seul le `get` est défini.

N'oubliez pas que la liste doit être initialisée avant d'être utilisée, sinon nous aurons une erreur. Le constructeur est un endroit approprié pour le faire :

```
public abstract class Compte
{
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public Compte()
    {
```

```
        listeOperations = new List<Operation>();
    }
}
```

La classe doit ensuite posséder des méthodes permettant de créditer ou de débiter le compte, ce qui donne :

```
public abstract class Compte
{
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public void Crediter(decimal montant)
    {
        Operation operation = new Operation { Montant = montant
            , TypeDeMouvement = Mouvement.Credit};
        listeOperations.Add(operation);
    }

    public void Crediter(decimal montant, Compte compte)
    {
        Crediter(montant);
        compte.Debiter(montant);
    }

    public void Debiter(decimal montant)
    {
        Operation operation = new Operation { Montant = montant
            , TypeDeMouvement = Mouvement.Debit };
        listeOperations.Add(operation);
    }

    public void Debiter(decimal montant, Compte compte)
    {
        Debiter(montant);
        compte.Crediter(montant);
    }
}
```

Le principe est de créer une opération et de l'ajouter à la liste des opérations. L'astuce ici consiste à réutiliser les méthodes de la classe pour écrire les autres formes des méthodes, ce qui simplifie le travail et facilitera les éventuelles futures opérations de maintenance.

Enfin, chaque classe dérivée de la classe `Compte` doit afficher un résumé du compte. Nous pouvons donc forcer ces classes à devoir implémenter cette méthode en utilisant une méthode abstraite :

```
public abstract class Compte
{
```

```
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public abstract void AfficherResume();
}
```

Voilà pour la classe `Compte`. En toute logique, c'est elle qui contient le plus de méthodes afin de factoriser le maximum de code commun dans la classe mère.

Passons à la classe `CompteEpargneEntreprise`, qui hérite de la classe `Compte`. Elle possède un taux d'abondement qui est défini à la création du compte. Il est donc ici intéressant de forcer le positionnement de ce taux lors de la création de la classe, c'est-à-dire en utilisant un constructeur avec un paramètre :

```
public class CompteEpargneEntreprise : Compte
{
    private double tauxAbondement;

    public CompteEpargneEntreprise(double taux)
    {
        tauxAbondement = taux;
    }
}
```

Ce taux est utilisé pour calculer le solde du compte en faisant la somme de toutes les opérations et en appliquant le taux ; ce qui revient à appeler le calcul du solde de la classe mère et à lui appliquer le taux. Nous substituons donc la propriété `Solde` et utilisons le calcul fait dans la classe `Compte` :

```
public class CompteEpargneEntreprise : Compte
{
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public override decimal Solde
    {
        get
        {
            return base.Solde * (decimal)(1 +  tauxAbondement);
        }
    }
}
```

Rien de plus simple, en utilisant le mot-clé `base` pour appeler le solde de la classe mère. Vous noterez également que nous avons eu besoin de caster le taux qui est un double afin de pouvoir le multiplier à un décimal.

Enfin, cette classe se doit de fournir une implémentation de la méthode `AfficherResume()` :

```
public class CompteEpargneEntreprise : Compte
```

```
{
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public override void AfficherResume()
    {
        Console.WriteLine("
            ####################################");
        Console.WriteLine("Compte épargne entreprise de " +
            Proprietaire);
        Console.WriteLine("\tSolde : " + Solde);
        Console.WriteLine("\tTaux : " + tauxAbondement);
        Console.WriteLine("\n\nOpérations :");
        foreach (Operation operation in listeOperations)
        {
            if (operation.TypeDeMouvement == Mouvement.Credit)
                Console.Write("\t+");
            else
                Console.Write("\t-");
            Console.WriteLine(operation.Montant);
        }
        Console.WriteLine("
            ####################################");
    }
}
```

Il s'agit d'une banale méthode d'affichage où nous parcourons la liste contenant les opérations et affichons le montant en fonction du type de mouvement.

Voilà pour la classe `CompteEpargneEntreprise`.

Il ne reste plus que la classe `CompteCourant` qui doit également dériver de `Compte` :

```
public class CompteCourant : Compte
{
}
```

Un compte courant doit posséder un découvert autorisé, défini à la création du compte courant. Nous utilisons comme avant un constructeur avec un paramètre :

```
public class CompteCourant : Compte
{
    private decimal decouvertAutorise;

    public CompteCourant(decimal decouvert)
    {
        decouvertAutorise = decouvert;
    }
}
```

Il ne restera plus qu'à fournir une implémentation de la méthode d'affichage du résumé :

```
public class CompteCourant : Compte
{
    // [...]
    // [Code précédent enlevé pour plus de lisibilité]
    // [...]

    public override void AfficherResume()
    {
        Console.WriteLine("*****************************************");
        Console.WriteLine("Compte courant de " + Proprietaire);
        Console.WriteLine("\tSolde : " + Solde);
        Console.WriteLine("\tDécouvert autorisé : " + decouvertAutorise);
        Console.WriteLine("\n\nOpérations :");
        foreach (Operation operation in listeOperations)
        {
            if (operation.TypeDeMouvement == Mouvement.Credit)
                Console.Write("\t+");
            else
                Console.Write("\t-");
            Console.WriteLine(operation.Montant);
        }
        Console.WriteLine("*****************************************");
    }
}
```

Voilà pour nos objets. Cela en fait un petit paquet. Mais ce n'est pas fini, il faut maintenant créer des comptes et faire des opérations.

L'énoncé consiste à faire les instanciations suivantes, depuis notre méthode `Main()` :

```
CompteCourant compteNicolas = new CompteCourant(2000) { Proprietaire = "Nicolas" };
CompteEpargneEntreprise compteEpargneNicolas = new CompteEpargneEntreprise(0.02) { Proprietaire = "Nicolas" };
CompteCourant compteJeremie = new CompteCourant(500) { Proprietaire = "Jérémie" };

compteNicolas.Crediter(100);
compteNicolas.Debiter(50);

compteEpargneNicolas.Crediter(20, compteNicolas);
compteEpargneNicolas.Crediter(100);

compteEpargneNicolas.Debiter(20, compteNicolas);

compteJeremie.Debiter(500);
compteJeremie.Debiter(200, compteNicolas);

```

```
Console.WriteLine("Solde compte courant de " + compteNicolas.
    Proprietaire + " : " + compteNicolas.Solde);
Console.WriteLine("Solde compte épargne de " +
    compteEpargneNicolas.Proprietaire + " : " +
    compteEpargneNicolas.Solde);
Console.WriteLine("Solde compte courant de " + compteJeremie.
    Proprietaire + " : " + compteJeremie.Solde);
Console.WriteLine("\n");
```

Rien d'extraordinaire.

Il ne reste plus qu'à afficher le résumé des deux comptes demandés :

```
Console.WriteLine("Résumé du compte de Nicolas");
compteNicolas.AfficherResume();
Console.WriteLine("\n");

Console.WriteLine("Résumé du compte épargne de Nicolas");
compteEpargneNicolas.AfficherResume();
Console.WriteLine("\n");
```

Nous en avons fini avec la première partie du TP.

Si vous avez bien compris comment construire des classes, comment les faire hériter entre elles et les interfaces, ce TP n'a pas dû vous poser trop de problèmes. Il existe bien sûr différentes solutions pour résoudre ce problème, de façon plus ou moins complexe.

Aller plus loin

Il y a plusieurs choses qui me dérangent dans ce code. La première est la répétition. Dans les deux méthodes `AfficherResume()` des classes respectives : `CompteCourant` et `CompteEpargneEntreprise`, on affiche les opérations de la même façon. Si jamais je dois modifier ce code (bug ou évolution), je devrais le faire dans les deux classes, au risque d'en oublier une. Il faut donc factoriser ce code. Il existe plusieurs solutions simples de factoriser ce code. La première est d'utiliser une méthode statique pour afficher la liste des opérations. Cette méthode pourrait être située dans une classe utilitaire qui prend en paramètre la liste des opérations :

```
public static class Helper
{
    public static void AfficheOperations(List<Operation>
        operations)
    {
        Console.WriteLine("\n\nOpérations :");
        foreach (Operation operation in operations)
        {
            if (operation.TypeDeMouvement == Mouvement.Credit)
                Console.Write("\t+");
            else
                Console.Write("\t-");
```

```
                Console.WriteLine(operation.Montant);
            }
        }
}
```

Il ne reste plus qu'à utiliser cette méthode statique depuis nos méthodes `AfficherResume()`, par exemple pour le compte épargne entreprise :

```
public override void AfficherResume()
{
    Console.WriteLine("#####################################");
    Console.WriteLine("Compte épargne entreprise de " +
        Proprietaire);
    Console.WriteLine("\tSolde : " + Solde);
    Console.WriteLine("\tTaux : " + tauxAbondement);
    Helper.AfficheOperations(listeOperations);
    Console.WriteLine("#####################################");
}
```

La deuxième solution est de créer la méthode `AfficheOperations()` dans la classe abstraite `Compte`, ce qui permet de s'affranchir de passer la liste en paramètre vu que la classe `Compte` la connaît déjà :

```
protected void AfficheOperations()
{
    Console.WriteLine("\n\nOpérations :");
    foreach (Operation operation in listeOperations)
    {
        if (operation.TypeDeMouvement == Mouvement.Credit)
            Console.Write("\t+");
        else
            Console.Write("\t-");
        Console.WriteLine(operation.Montant);
    }
}
```

La méthode a tout intérêt à être déclarée en `protected`, afin qu'elle puisse servir aux classes filles mais pas à l'extérieur. Elle s'utilisera de cette façon, par exemple dans la classe `CompteEpargneEntreprise` :

```
public override void AfficherResume()
{
    Console.WriteLine("#####################################");
    Console.WriteLine("Compte épargne entreprise de " +
        Proprietaire);
    Console.WriteLine("\tSolde : " + Solde);
    Console.WriteLine("\tTaux : " + tauxAbondement);
    AfficheOperations();
    Console.WriteLine("#####################################");
}
```

Dès que l'on peut factoriser du code, il ne faut pas hésiter. Si nous avons demain besoin de créer un nouveau type de compte, nous serons ravis de pouvoir nous servir de méthodes toutes prêtes nous simplifiant le travail.

Il pourrait être intéressant également d'encapsuler l'enregistrement d'une opération dans une méthode. Ça permettrait de moins se répéter (même si ici, le code est vraiment petit) mais surtout de séparer la logique d'enregistrement d'une opération afin que cela soit plus facile ultérieurement à modifier, maintenir ou complexifier. Par exemple, via une méthode :

```
private void EnregistrerOperation(Mouvement typeDeMouvement,
    decimal montant)
{
    Operation operation = new Operation { Montant = montant,
        TypeDeMouvement = typeDeMouvement};
    listeOperations.Add(operation);
}
```

Cela permet également de faire en sorte que le type de mouvement soit un paramètre de la méthode.

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 729727

Deuxième partie du TP

Nous voici maintenant dans la deuxième partie du TP. La banque souhaite proposer un nouveau type de compte, le livret ToutBénéf. Dans cette banque, le livret ToutBénéf fonctionne comme le compte épargne entreprise. C'est-à-dire qu'il accepte un taux en paramètres et applique ce taux au moment de la restitution du solde.

La première idée qui vient à l'esprit est de créer une classe `LivretToutBenef` qui hérite de `CompteEpargneEntreprise`. Mais ceci pose un problème si jamais le compte épargne entreprise doit évoluer, et c'est justement ce que le directeur de la banque vient de me confier. Donc, il vous interdit à juste titre d'hériter de ses fonctionnalités.

Ce que vous allez donc faire ici, c'est de considérer que le fait qu'un compte puisse faire des bénéfices soit en fait un comportement qui est fourni au moment où on instancie un compte. Il existe plusieurs comportements dont on doit fournir les implémentations :

– le comportement de bénéfice à taux fixe ;
– le comportement de bénéfice aléatoire ;
– le comportement où il n'y a aucun bénéfice.

Chaque comportement est une classe qui respecte le contrat suivant :

```
public interface ICalculateurDeBenefice
{
    decimal CalculeBenefice(decimal solde);
```

```
    double Taux { get; }
}
```

Écrivez donc ces trois classes de comportement ainsi que le livret `ToutBénéf` qui possède un taux fixe de 2.75% et qui a été crédité une première fois de 800 € et une seconde fois de 200 €. Affichez enfin son résumé qui devra tenir compte du taux du calculateur de bénéfice. Réécrivez ensuite la classe `CompteCourant` de manière à ce qu'elle ait un comportement où il n'y a pas de bénéfice. Enfin, la classe `CompteEpargneEntreprise` subira une évolution pour fonctionner avec un comportement de bénéfice aléatoire (tiré entre 0 et 1).

C'est parti.

Correction

Ici c'est un peu plus compliqué. Vous n'êtes sans doute pas complètement familiers avec la notion d'interface, aussi avant de vous donner la correction, je vais vous donner quelques pistes. Le fait d'avoir un comportement est finalement très simple. Il suffit d'avoir un membre privé dans la classe `LivretToutBenef` du type de l'interface. Ce membre privé sera affecté à la valeur passée en paramètre du constructeur. C'est-à-dire :

```
public class LivretToutBenef : Compte
{
    private ICalculateurDeBenefice calculateurDeBenefice;

    public LivretToutBenef(ICalculateurDeBenefice calculateur)
    {
        calculateurDeBenefice = calculateur;
    }
}
```

Désormais, les opérations devront se faire avec cette variable, `calculateurDeBenefice`. On aura également besoin d'instancier au préalable le comportement voulu et de le passer en paramètre du constructeur. Retournez donc tenter de réaliser ce TP avant de voir la suite de la correction !

C'est fait ? Alors voici ma correction. Nous avions donc cette interface imposée :

```
public interface ICalculateurDeBenefice
{
    decimal CalculeBenefice(decimal solde);
    double Taux { get; }
}
```

Nous avons besoin d'écrire plusieurs classes qui implémentent cette interface. La première est la classe de bénéfice à taux fixe :

```
public class BeneficeATauxFixe : ICalculateurDeBenefice
{
    private double taux;
```

```

    public BeneficeATauxFixe(double tauxFixe)
    {
        taux = tauxFixe;
    }

    public decimal CalculeBenefice(decimal solde)
    {
        return solde * (decimal)(1 + taux);
    }

    public double Taux
    {
        get
        {
            return taux;
        }
    }
}
```

Nous avons dit qu'elle devait prendre un taux en paramètre ; le constructeur est l'endroit indiqué pour cela. Ensuite, la méthode de calcul est très simple, il suffit d'appliquer la formule au solde. Enfin, la propriété retourne le taux.

La classe suivante est la classe de bénéfice aléatoire. Là, pas besoin de paramètres, il suffit de tirer le nombre aléatoire dans le constructeur grâce à la méthode `NextDouble()`, ce qui donne :

```
public class BeneficeAleatoire : ICalculateurDeBenefice
{
    private double taux;
    private Random random;

    public BeneficeAleatoire()
    {
        random = new Random();
        taux = random.NextDouble();
    }

    public decimal CalculeBenefice(decimal solde)
    {
        return solde * (decimal)(1 + taux);
    }

    public double Taux
    {
        get
        {
            return taux;
        }
    }
```

```
}
```

Enfin, il faudra créer la classe avec aucun bénéfice :

```
public class AucunBenefice : ICalculateurDeBenefice
{
    public decimal CalculeBenefice(decimal solde)
    {
        return solde;
    }

    public double Taux
    {
        get
        {
            return 0;
        }
    }
}
```

Rien de sorcier.

Là où ça se complique un peu, c'est pour la classe `LivretToutBenef`. Elle doit bien sûr dériver de la classe de base `Compte` et posséder un membre privé de type `ICalculateurDeBenefice` :

```
public class LivretToutBenef : Compte
{
    private ICalculateurDeBenefice calculateurDeBenefice;

    public LivretToutBenef(ICalculateurDeBenefice calculateur)
    {
        calculateurDeBenefice = calculateur;
    }

    public override decimal Solde
    {
        get
        {
            decimal solde = base.Solde;
            return solde + calculateurDeBenefice.
                CalculeBenefice(solde);
        }
    }

    public override void AfficherResume()
    {
        Console.WriteLine("^^^^^^^^^^^^^^^^^");
        Console.WriteLine("Livret ToutBénéf de " + Proprietaire
            );
        Console.WriteLine("\tSolde : " + Solde);
```

```
            Console.WriteLine("\tTaux : " + calculateurDeBenefice.
                Taux);
            AfficheOperations();
            Console.WriteLine("^^^^^^^^^^^^^^^^^");
        }
}
```

Dans le constructeur, la variable est initialisée avec un objet de type `ICalculateurDeBenefice`. Ensuite, pour calculer le solde, il suffit d'appeler la méthode `CalculeBenefice` avec le solde de base en paramètre. De même, pour faire apparaître le taux dans le résumé, on pourra utiliser la propriété `Taux` de l'interface. Il ne reste qu'à souscrire à un Livret ToutBénéf en lui passant un objet de bénéfice à taux fixe en paramètre du constructeur, et à faire les opérations bancaires demandées. Ce qui donne :

```
ICalculateurDeBenefice beneficeATauxFixe = new
    BeneficeATauxFixe(0.275);
LivretToutBenef livretToutBenefNicolas = new LivretToutBenef(
    beneficeATauxFixe);
livretToutBenefNicolas.Crediter(800);
livretToutBenefNicolas.Crediter(200);

Console.WriteLine("Résumé du livret ToutBénéf");
livretToutBenefNicolas.AfficherResume();
Console.WriteLine("\n");
```

Vous aurez donc :

```
Résumé du livret ToutBénéf
^^^^^^^^^^^^^^^^^
Livret ToutBénéf de
        Solde : 2275,000
        Taux : 0,275

Opérations :
        +800
        +200
^^^^^^^^^^^^^^^^^
```

Maintenant, nous devons adapter la classe `CompteEpargneEntreprise` pour qu'elle puisse fonctionner avec un comportement. Cela devient tout simple et se rapproche beaucoup du livret ToutBénéf :

```
public class CompteEpargneEntreprise : Compte
{
    private ICalculateurDeBenefice calculateurDeBenefice;

    public CompteEpargneEntreprise(ICalculateurDeBenefice
        calculateur)
    {
```

```
            calculateurDeBenefice = calculateur;
        }

        public override decimal Solde
        {
            get
            {
                decimal solde = base.Solde;
                return solde + calculateurDeBenefice.
                    CalculeBenefice(solde);
            }
        }

        public override void AfficherResume()
        {
            Console.WriteLine("
                ####################################");
            Console.WriteLine("Compte épargne entreprise de " +
                Proprietaire);
            Console.WriteLine("\tSolde : " + Solde);
            Console.WriteLine("\tTaux : " + calculateurDeBenefice.
                Taux);
            AfficheOperations();
            Console.WriteLine("
                ####################################");
        }
    }
```

Il faut maintenant changer l'instanciation de l'objet `CompteEpargneEntreprise` en lui passant en paramètre un objet de type bénéfice aléatoire :

```
ICalculateurDeBenefice beneficeAleatoire = new
    BeneficeAleatoire();
CompteEpargneEntreprise compteEpargneNicolas = new
    CompteEpargneEntreprise(beneficeAleatoire) { Proprietaire =
    "Nicolas" };
```

Il reste le compte courant qui suit le même principe :

```
public class CompteCourant : Compte
{
    private decimal decouvertAutorise;
    private ICalculateurDeBenefice calculateurDeBenefice;

    public CompteCourant(decimal decouvert,
        ICalculateurDeBenefice calculateur)
    {
        decouvertAutorise = decouvert;
        calculateurDeBenefice = calculateur;
    }

```

```
    public override decimal Solde
    {
        get
        {
            decimal solde = base.Solde;
            return solde + calculateurDeBenefice.CalculeBenefice(solde);
        }
    }

    public override void AfficherResume()
    {
        Console.WriteLine("*******************************************");
        Console.WriteLine("Compte courant de " + Proprietaire);
        Console.WriteLine("\tSolde : " + Solde);
        Console.WriteLine("\tDécouvert autorisé : " + decouvertAutorise);
        Console.WriteLine("\tTaux : " + calculateurDeBenefice.Taux);
        AfficheOperations();
        Console.WriteLine("*******************************************");
    }
}
```

Que l'on pourra instancier avec un objet de type aucun bénéfice :

```
ICalculateurDeBenefice aucunBenefice = new AucunBenefice();
CompteCourant compteNicolas = new CompteCourant(2000, aucunBenefice) { Proprietaire = "Nicolas" };
```

Et voilà ! L'avantage ici est d'avoir séparé les responsabilités dans différentes classes. Si jamais nous créons un nouveau compte qui est rémunéré grâce à un bénéfice à taux fixe, il suffira de réutiliser ce comportement et le tour est joué.

À noter que les trois calculs de la propriété `Solde` sont identiques, il pourrait être judicieux de le factoriser dans la classe mère `Compte`. Ceci implique que la classe mère possède elle-même le membre protégé du type de l'interface.

Voilà pour ce TP. J'espère que vous aurez réussi avec brio la création de toutes les classes et que vous ne vous êtes pas perdus dans les mots-clés. Vous verrez que vous aurez très souvent besoin d'écrire des classes dans ce genre afin de créer une application. C'est un élément indispensable du C# qu'il est primordial de maîtriser.

N'hésitez pas à faire des variations sur ce TP ou à créer d'autres petits programmes simples vous permettant de vous entraîner !

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 817413

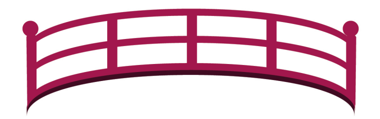
Chapitre 25

Mode de passage des paramètres à une méthode

Difficulté : 

Dans les chapitres précédents, nous avons décrit comment on pouvait passer des paramètres à une méthode. Nous avons également vu que les types du framework .NET pouvaient être des types valeur ou des types référence. Ceci influence la façon dont sont traités les paramètres d'une méthode. Nous allons ici préciser un peu ce fonctionnement.

Dans ce chapitre, je vais illustrer mes propos en utilisant des méthodes statiques écrites dans la classe `Program`, générée par Visual C# Express. Le but est de simplifier l'écriture et de ne pas s'encombrer d'objets inutiles. Évidemment, tout ce que nous allons voir fonctionne également de la même façon avec les méthodes non statiques présentes dans des objets.



Passage de paramètres par valeur

Au tout début de ce livre, nous avons appelé des méthodes en passant des types simples (`int`, `string`, etc.). Nous avons vu qu'il s'agissait de types valeur qui possèdent directement la valeur dans la variable.

Lorsque nous passons des types valeur en paramètres d'une méthode, nous utilisons un passage de paramètre par valeur.

Décortiquons l'exemple suivant :

```
static void Main(string[] args)
{
    int age = 30;
    Doubler(age);
    Console.WriteLine(age);
}

public static void Doubler(int valeur)
{
    valeur = valeur * 2;
    Console.WriteLine(valeur);
}
```

Nous déclarons dans la méthode `Main()` une variable `age`, de type entier, à laquelle nous affectons la valeur 30. Nous appelons ensuite la méthode `Doubler()` en lui passant cette variable en paramètre.

Ce qu'il se passe c'est que le compilateur fait une copie de la valeur de la variable `age` pour la mettre dans la variable `valeur` de la méthode `Doubler()`. La variable `valeur` a une portée égale au corps de la méthode `Doubler()`.

Nous modifions ensuite la valeur de la variable `valeur` en la multipliant par deux.

Étant donné que la variable `valeur` a reçu une copie de la variable `age`, c'est-à-dire que le compilateur a dupliqué la valeur 30, le fait de modifier la variable `valeur` ne change en rien la valeur de la variable `age` :

```
60
30
```

Nous passons 30 à la méthode `Doubler()` qui calcule le double de la variable `valeur`. On affiche 60. Lorsqu'on revient dans la méthode `Main()`, nous retrouvons la valeur initiale de la variable `age`. Elle n'a bien sûr pas été modifiée, car la méthode `Doubler()` a travaillé sur une copie.

Rappelez-vous que ceci est possible car les types intégrés sont facilement copiables, car peu évolués.

Passage de paramètres en mise à jour



Oui, mais si je veux modifier la valeur ?

Pour pouvoir modifier la valeur du paramètre passé, il faut indiquer que la variable est en mode « mise à jour ». Cela se fait grâce au mot-clé « `ref` » que nous pourrons utiliser ainsi :

```
static void Main(string[] args)
{
    int age = 30;
    Doubler(ref age);
    Console.WriteLine(age);
}

public static void Doubler(ref int valeur)
{
    valeur = valeur * 2;
}
```

Comme on peut s'en douter, ce code affiche 60.

Le mot-clé `ref` s'utilise avant la définition du paramètre dans la méthode. Cela implique qu'il soit également utilisé au moment d'appeler la méthode.

Vous aurez remarqué que le mot-clé utilisé est `ref` et ressemble beaucoup au mot « **référence** ». Ce n'est évidemment pas un hasard ! En fait, avec ce mot-clé nous demandons au compilateur de passer en paramètre une référence vers la variable `age` plutôt que d'en faire une copie. Ainsi, la méthode `Doubler()` récupère une référence et la variable `valeur` référence alors la même valeur que la variable `age`. Ceci implique que toute modification de la valeur référencée provoque un changement sur la variable source puisque les variables référencent la même valeur.

Voilà pourquoi la variable est modifiée après le passage dans la méthode `Doubler()`.

Bien sûr, il aurait tout à fait été possible de faire en sorte que la méthode `Doubler()` renvoie un entier contenant la valeur passée en paramètre multipliée par 2 :

```
static void Main(string[] args)
{
    int age = 30;
    age = Doubler(age);
    Console.WriteLine(age);
}

public static int Doubler(int valeur)
{
    return valeur * 2;
}
```

C'est ce que nous avons toujours fait auparavant. Voici maintenant une autre façon de faire qui peut être bien utile quand il y a plus d'une valeur à renvoyer.

Passage des objets par référence

C'est aussi comme ça que cela fonctionne pour les objets. Nous avons vu que les variables qui stockent des objets possèdent en fait la référence de l'objet. Le fait de passer un objet à une méthode équivaut à passer la référence de l'objet en paramètre. Ainsi, c'est comme si on utilisait le mot-clé `ref` implicitement.

Le code suivant :

```
static void Main(string[] args)
{
    Voiture voiture = new Voiture { Couleur = "Grise" };
    Repeindre(voiture);
    Console.WriteLine(voiture.Couleur);
}

public static void Repeindre(Voiture voiture)
{
    voiture.Couleur = "Bleue";
}
```

va donc créer un objet `Voiture` et le passer en paramètre à la méthode `Repeindre()`. Comme `Voiture` est un type référence, il n'y a pas de duplication de l'objet et une référence est passée à la méthode.

Lorsque nous modifions la propriété `Couleur` de la voiture, nous modifions bien le même objet que celui présent dans la méthode `Main()` :

```
Bleue
```

Il est à noter quand même que la variable `voiture` de la méthode `Repeindre` est une copie de la variable `voiture` de la méthode `Main()`. Elles contiennent toutes les deux une référence vers l'objet de type `Voiture`. Cela veut dire que l'on accède bien au même objet, d'où le résultat, mais que les deux variables sont indépendantes. Si nous modifions directement la variable, avec par exemple :

```
static void Main(string[] args)
{
    Voiture voiture = new Voiture { Couleur = "Grise" };
    Repeindre(voiture);
    Console.WriteLine(voiture.Couleur);
}

public static void Repeindre(Voiture voiture)
{
    voiture.Couleur = "Bleue";
```

```
    voiture = null;
}
```

alors, ce code continuera à fonctionner, car la variable `voiture` de la méthode `Main()` ne vaut pas `null`. Le fait de modifier la variable de la méthode `Repeindre`, qui est une copie, n'affecte en rien la variable de la méthode `Main()`.

Par contre, elle est affectée si nous utilisons le mot-clé `ref`. Par exemple le code suivant :

```
static void Main(string[] args)
{
    Voiture voiture = new Voiture { Couleur = "Grise" };
    Repeindre(ref voiture);
    Console.WriteLine(voiture.Couleur);
}

public static void Repeindre(ref Voiture voiture)
{
    voiture.Couleur = "Bleue";
    voiture = null;
}
```

provoquera une erreur. En effet, cette fois-ci, c'est bien la référence qui est passée à nulle et pas une copie de la variable contenant la référence...

Une différence subtile !

Passage de paramètres en sortie

Enfin, le dernier mode de passage est le passage de paramètres en sortie. Il permet de faire en sorte qu'une méthode force l'initialisation d'une variable et que l'appelant récupère la valeur initialisée. Nous avons déjà vu ce mode de passage quand nous avons étudié les conversions. Souvenez-vous de ce code :

```
string chaine = "1234";
int nombre;
if (int.TryParse(chaine, out nombre))
    Console.WriteLine(nombre);
else
    Console.WriteLine("Conversion impossible");
```

La méthode `TryParse` permet de tenter la conversion d'une chaîne. Elle renvoie vrai ou faux en fonction du résultat de la conversion et met à jour l'entier qui est passé en paramètre en utilisant le mot-clé `out`. Si la conversion réussit, alors l'entier `nombre` est initialisé avec la valeur de la conversion, calculée dans la méthode `TryParse`. Nous pouvons utiliser ensuite la variable `nombre` car le mot-clé `out` garantit que la variable sera initialisée dans la méthode.

En effet, si nous prenons l'exemple suivant :

```
static void Main(string[] args)
```

```
{
    int age = 30;
    int ageDouble;
    Doubler(age, out ageDouble);
}

public static void Doubler(int age, out int resultat)
{
}
```

Nous aurons une erreur de compilation :

```
Le paramètre de sortie 'resultat' doit être assigné avant que le
    contrôle quitte la méthode actuelle
```

En effet, il faut absolument que la variable `resultat` qui est marquée en sortie ait une valeur avant de pouvoir sortir de la méthode.

Nous pourrons corriger cet exemple avec :

```
static void Main(string[] args)
{
    int age = 30;
    int ageDouble;
    Doubler(age, out ageDouble);
}

public static void Doubler(int age, out int resultat)
{
    resultat = age * 2;
}
```

Après l'appel de la méthode `Doubler()`, `ageDouble` vaudra donc 60.



Oui, mais quel intérêt par rapport à un `return` normal ?

Ici, aucun. C'est pertinent quand nous souhaitons renvoyer plusieurs valeurs, comme c'est le cas pour la méthode `TryParse` qui renvoie le résultat de la conversion et si la conversion s'est bien passée.

Bien sûr, si l'on n'aime pas trop le mot-clé `out`, il est toujours possible de créer un objet contenant deux valeurs que l'on retournera à l'appelant, par exemple :

```
public class Program
{
    static void Main(string[] args)
    {
        string nombre = "1234";
        Resultat resultat = TryParse(nombre);
```

```
        if (resultat.ConversionOk)
            Console.WriteLine(resultat.Valeur);
    }

    public static Resultat TryParse(string chaine)
    {
        Resultat resultat = new Resultat();
        int valeur;
        resultat.ConversionOk = int.TryParse(chaine, out valeur
            );
        resultat.Valeur = valeur;
        return resultat;
    }

}

public class Resultat
{
    public bool ConversionOk { get; set; }
    public int Valeur { get; set; }
}
```

Ici, notre méthode `TryParse` renvoie un objet `Resultat` qui contient les deux valeurs résultantes de la conversion.

En résumé

– Le type d'une variable passée en paramètres d'une méthode influence la façon dont elle est traitée.
– Un passage par valeur effectue une copie de la valeur de la variable et la met dans la variable de la méthode.
– Un passage par référence effectue une copie de la référence mais continue de pointer sur le même objet.
– On utilise le mot-clé `ref` pour passer une variable de type valeur à une méthode afin de la modifier.

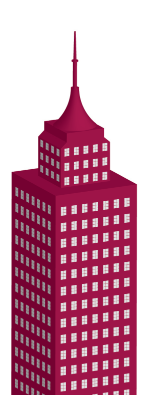
Chapitre 26

Les structures

Difficulté : 

Nous allons aborder dans ce chapitre les structures, qui sont une nouvelle sorte d'objets que nous pouvons créer dans des applications C#. Les structures sont presque comme des classes. Elles permettent également de créer des objets, possèdent des variables ou propriétés, des méthodes et même un constructeur, mais avec quelques subtiles différences...

Découvrons-les dès à présent !



Une structure est presque une classe

Une des premières différences entre la `classe` et la **structure** est la façon dont .NET gère ces deux objets. Nous avons vu que les classes étaient des types référence. Les variables de type référence ne possèdent donc pas la valeur de l'objet mais une référence vers cet objet en mémoire. La structure, quant à elle, est un type valeur et contient donc directement la valeur de l'objet.

Une autre différence est que la structure, bien qu'étant un objet, ne peut pas utiliser les principes d'héritage. On ne peut donc pas hériter d'une structure et une structure ne peut pas hériter des comportements d'un objet.

À quoi sert une structure ?

Les structures vont être utiles pour stocker de petits objets amenés à être souvent manipulés, comme les `int` ou les `bool` que nous avons déjà vus.

La raison tient en un seul mot : **performance**.

Étant gérées en mémoire différemment, les structures sont optimisées pour améliorer les performances des petits objets. Comme il n'y a pas de référence, on utilisera directement l'objet sans aller le chercher via sa référence. On gagne donc un peu de temps lorsqu'on a besoin de manipuler ces données.

C'est tout à fait pertinent pour des programmes où la vitesse est déterminante, comme les jeux vidéo.



Donc dans ce cas, autant utiliser tout le temps des structures, non ?

Eh bien non, déjà vous vous priveriez de tous les mécanismes d'héritage que nous avons vus. Ensuite, si nous surchargeons trop la mémoire avec des structures, l'optimisation prévue par .NET risque de se retourner contre nous et notre application pourrait être plus lente que si nous avions utilisé des classes.

D'une manière générale, et à moins de savoir exactement ce que vous faites ou d'avoir mesuré les performances, vous allez utiliser plus généralement les classes que les structures. Vous pouvez à ce sujet lire les recommandations de Microsoft, disponibles grâce au code web suivant :

▷ Documentation Microsoft
Code web : 520338
.

Créer une structure

Pour créer une structure, nous utiliserons le mot-clé `struct`, comme nous avions utilisé le mot-clé `class` pour créer une classe :

```
public struct Personne
{
    public string Prenom { get; set; }
    public int Age { get; set; }
}
```

Pour instancier cette structure, nous pourrons utiliser le mot-clé `new`, comme pour les classes. La différence est que la variable sera un type valeur, avec les conséquences que ce type impose en matière de gestion en mémoire ou de passages par paramètres :

```
Personne nicolas = new Personne() { Prenom = "Nicolas", Age =
    30 };
Console.WriteLine(nicolas.Prenom + " a " + nicolas.Age + " ans"
    );
```

Comme nous avons dit, il est impossible qu'une structure hérite d'une autre structure ou d'un objet ; sauf bien sûr du fameux type de base `object`, pour qui c'est automatique. Une structure hérite donc des quelques méthodes d'`Object` (comme `ToString()`) que nous pouvons éventuellement spécialiser :

```
public struct Personne
{
    public string Prenom { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return Prenom + " a " + Age + " ans";
    }
}
```

Et nous pourrons écrire :

```
Personne nicolas = new Personne() { Prenom = "Nicolas", Age =
    30 };
Console.WriteLine(nicolas.ToString());
```

Qui renverra :

```
Nicolas a 30 ans
```

Comme pour les classes, il est possible d'avoir des constructeurs sur une structure, à l'exception du constructeur par défaut qui est interdit.

Aussi le code suivant :

```
public struct Personne
{
    public Personne()
    {
    }
}
```

provoquera l'erreur de compilation suivante :

```
Les structures ne peuvent pas contenir de constructeurs exempts
   de paramètres explicites
```

Par contre, les autres formes des constructeurs sont possibles, comme :

```
public struct Personne
{
    private int age;
    public Personne(int agePersonne)
    {
        age = agePersonne;
    }
}
```

Ce constructeur s'utilisera comme pour une classe :

```
Personne nicolas = new Personne(30);
```



Attention, si vous tentez d'utiliser des propriétés ou des méthodes dans le constructeur d'une structure, vous allez avoir un problème !

Par exemple le code suivant :

```
public struct Personne
{
    private int age;
    public Personne(int agePersonne)
    {
        AffecteAge(agePersonne);
    }

    private void AffecteAge(int agePersonne)
    {
        age = agePersonne;
    }
}
```

provoquera les erreurs de compilation suivantes :

```
L'objet 'this' ne peut pas être utilisé avant que tous ses
   champs soient assignés
```

```
Le champ 'MaPremiereApplication.Personne.age' doit être
    totalement assigné avant que le contrôle soit retourné à l'
    appelant
```

Alors qu'avec une classe, ce code serait tout à fait correct.

Pour corriger ceci, il faut absolument initialiser tous les champs avant de faire quoi que ce soit avec l'objet, comme l'indique l'erreur.

Nous pourrons par exemple faire :

```
public struct Personne
{
    private int age;
    public Personne(int agePersonne)
    {
        age = 0;
        AffecteAge(agePersonne);
    }

    private void AffecteAge(int agePersonne)
    {
        age = agePersonne;
    }
}
```

Ce qui peut sembler tout à fait inutile dans ce cas-là. Mais comme le compilateur fait certaines vérifications, il sera impossible de compiler un code de ce genre sans que toutes les variables soient initialisées explicitement.

Par contre, vous aurez un souci si vous utilisez des propriétés automatiques. Si nous tentons de faire :

```
public struct Personne
{
    public int Age { get; set; }
    public Personne(int agePersonne)
    {
        Age = agePersonne;
    }
}
```

nous nous retrouverons avec la même erreur de compilation. Pour la corriger, il faudra appeler le constructeur par défaut de la structure qui va permettre d'initialiser toutes les variables de la classe :

```
public struct Personne
{
    public int Age { get; set; }
    public Personne(int agePersonne) : this()
    {
        Age = agePersonne;
```

```
    }
}
```

Cela se fait comme pour les classes, en utilisant le mot-clé `this` suivi de parenthèses, qui permettra d'appeler le constructeur par défaut.

Rappelez-vous que le constructeur par défaut s'occupe d'initialiser toutes les variables d'une classe ou d'une structure.

Passage de structures en paramètres

Comme il s'agit d'un type valeur, à chaque fois que nous passerons une structure en paramètres d'une méthode, une copie de l'objet sera faite.

Ainsi, tout à fait logiquement, le code suivant :

```
static void Main(string[] args)
{
    Personne nicolas = new Personne() { Age = 30 };
    FaitVieillir(nicolas);
    Console.WriteLine(nicolas.Age);
}

private static void FaitVieillir(Personne personne)
{
    personne.Age++;
}
```

affichera 30, bien que nous modifions l'âge de la personne dans la méthode.

Comme nous l'avons déjà vu, la méthode travaille sur une copie de la structure.

Cela veut bien sûr dire que si nous souhaitons modifier une structure à partir d'une méthode, nous devrons utiliser le mot-clé `ref` :

```
static void Main(string[] args)
{
    Personne nicolas = new Personne() { Age = 30 };
    FaitVieillir(ref nicolas);
    Console.WriteLine(nicolas.Age);
}

private static void FaitVieillir(ref Personne personne)
{
    personne.Age++;
}
```

Ceci vaut pour tous les types valeur.



Prenez quand même garde. Si la structure est très grosse, le fait d'en faire une copie à chaque utilisation de méthode risque d'être particulièrement chronophage et pourra perturber les performances de votre application.

D'autres structures ?

Vous l'aurez peut-être deviné, mais les entiers que nous avons déjà vus et que nous utilisions grâce au mot-clé `int` sont en fait des structures.

Étant très souvent utilisés et n'ayant pas non plus énormément de choses à stocker, ils sont créés en tant que structures et sont optimisés par .NET pour que nos applications s'exécutent de façon optimale ; ce qui est un choix tout à fait pertinent.

C'est le cas également pour les `bool`, les `double`, etc.

À noter en revanche que d'autres objets, comme la classe `String`, sont bel et bien des classes.

D'une manière générale, vous allez créer peu de structures en tant que débutant. Il sera plus judicieux de créer des classes dès que vous en avez besoin. En effet, plus vos objets sont gros et plus ils auront intérêt à être des classes pour éviter d'être recopiés à chaque utilisation.

L'utilisation de structures pourra se révéler pertinente dans des situations bien précises, mais en général, il faut bien maîtriser les rouages du framework .NET pour que les bénéfices de leur utilisation se fassent ressentir.

Dans tous les cas, il sera important de mesurer (avec par exemple des outils de profilage) le gain de temps avant de mettre des structures partout.

En résumé

- Les structures sont des types valeur qui sont optimisés par le framework .NET.
- Une structure est un objet qui ressemble beaucoup à une classe, mais qui possède des restrictions.
- Les structures possèdent des propriétés, des méthodes, des constructeurs, etc.
- Il n'est pas possible d'utiliser l'héritage avec les structures.

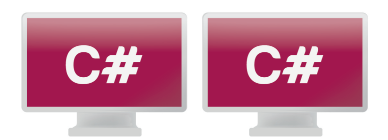
Chapitre 27

Les génériques

Difficulté : 

Les génériques sont une fonctionnalité du framework .NET apparus avec la version 2. Vous vous en souvenez peut-être, nous avons cité le mot dans le chapitre sur les tableaux et les listes. Ils permettent de créer des méthodes ou des classes qui sont indépendantes d'un type. Il est très important de connaître leur fonctionnement car c'est un mécanisme clé qui permet de faire beaucoup de choses, notamment en termes de réutilisabilité et d'amélioration des performances.

N'oubliez pas vos tubes d'aspirine et voyons à présent de quoi il retourne !



Qu'est-ce que les génériques ?

Avec les génériques, vous pouvez créer des méthodes ou des classes qui sont indépendantes d'un type. On les appellera des méthodes génériques et des types génériques.

Nous en avons déjà utilisé, rappelez-vous, avec la liste.

La liste est une classe comme nous en avons déjà vu plein, sauf qu'elle a la capacité d'être utilisée avec n'importe quel autre type, comme les entiers, les chaînes de caractères, les voitures...

Cela permet d'éviter de devoir créer une classe `ListeInt`, une classe `ListeString`, une classe `ListeVoiture`, etc. On pourra utiliser cette classe avec tous les types grâce aux chevrons :

```
List<string> listeChaine = new List<string>();
List<int> listeEntier = new List<int>();
List<Voiture> listeVoiture = new List<Voiture>();
```

Nous indiquons entre les chevrons le type qui sera utilisé avec le type générique.



Oui mais, si nous voulons pouvoir mettre n'importe quel type d'objet dans une liste, il suffirait de créer une `ListeObject` ? Puisque tous les objets dérivent d'`object`...

En fait, c'est le choix qui avait été fait dans la toute première version de .NET. On utilisait l'objet `ArrayList` qui possède une méthode `Add` prenant en paramètre un `object`. Cela fonctionne. Sauf que nous nous trouvions face à des limitations.

Premièrement, nous pouvions mélanger n'importe quel type d'objet dans la liste, des entiers, des voitures, des chiens, etc. Cela devenait une classe fourre-tout et nous ne savions jamais ce qu'il y avait dans la liste.

Deuxièmement, même si nous savions qu'il n'y avait que des entiers dans la liste, nous étions obligés de les traiter en tant qu'`object` et donc d'utiliser le boxing et l'unboxing pour mettre les objets dans la liste ou pour les récupérer.

Cela engendrait donc confusion et perte de performance. Grâce aux génériques, il devenait donc possible de créer des listes de n'importe quel type avec la garantie de savoir exactement quel type nous allions récupérer dans la liste.

Les types génériques du framework .NET

Le framework .NET possède beaucoup de classes et d'interfaces génériques, notamment dans l'espace de nom `System.Collections.Generic`.

La liste est la classe générique que vous utiliserez sûrement le plus. Mais beaucoup d'autres sont à votre disposition. Citons par exemple la classe `Queue<>` qui permet de gérer une file d'attente style FIFO [1] :

1. Acronyme de l'expression anglaise *First In, First Out* : premier entré, premier sorti.

```
Queue<int> file = new Queue<int>();
file.Enqueue(3);
file.Enqueue(1);
int valeur = file.Dequeue(); // valeur contient 3
valeur = file.Dequeue(); // valeur contient 1
```

Citons encore le dictionnaire d'éléments qui est une espèce d'annuaire où l'on accède aux éléments grâce à une clé :

```
Dictionary<string, Personne> annuaire = new Dictionary<string,
    Personne>();
annuaire.Add("06 01 02 03 04", new Personne { Prenom = "Nicolas
    "});
annuaire.Add("06 06 06 06 06", new Personne { Prenom = "Jeremie
    " });

Personne p = annuaire["06 06 06 06 06"]; // p contient la
    propriété Prenom valant Jeremie
```

Loin de moi l'idée de vous énumérer toutes les collections génériques du framework .NET ; le but est de vous montrer rapidement qu'il existe beaucoup de classes génériques dans le framework .NET.

Créer une méthode générique

Nous commençons à cerner l'intérêt des génériques. Sachez qu'il est bien sûr possible de créer vos propres classes génériques ou vos propres méthodes.

Commençons par les méthodes, ça sera plus simple. Il est globalement intéressant d'utiliser un type générique partout où nous pourrions avoir un `object`.

Dans la première partie, nous avions créé une méthode `AfficheRepresentation()` qui prenait un objet en paramètre, ce qui pourrait être :

```
public static class Afficheur
{
    public static void Affiche(object o)
    {
        Console.WriteLine("Afficheur d'objet :");
        Console.WriteLine("\tType : " + o.GetType());
        Console.WriteLine("\tReprésentation : " + o.ToString())
            ;
    }
}
```

Nous avons ici utilisé une classe statique permettant d'afficher le type d'un objet et sa représentation. Nous pouvons l'utiliser ainsi :

```
int i = 5;
double d = 9.5;
```

```
string s = "abcd";
Voiture v = new Voiture();

Afficheur.Affiche(i);
Afficheur.Affiche(d);
Afficheur.Affiche(s);
Afficheur.Affiche(v);
```

Rappelez-vous, chaque fois qu'on passe dans cette méthode, l'objet est boxé en type `object` quand il s'agit d'un type valeur.

Nous pouvons améliorer cette méthode en créant une méthode générique. Regardons ce code :

```
public static class Afficheur
{
    public static void Affiche<T>(T a)
    {
        Console.WriteLine("Afficheur d'objet :");
        Console.WriteLine("\tType : " + a.GetType());
        Console.WriteLine("\tReprésentation : " + a.ToString())
            ;
    }
}
```

Cette méthode fait exactement la même chose mais avec les génériques.

Dans un premier temps, la méthode annonce qu'elle va utiliser un type générique représenté par la lettre « T » entre chevrons.



Il est conventionnel que les types génériques soient utilisés avec « T ».

Cela signifie que tout type utilisé dans cette méthode, déclaré avec T, sera du type passé à la méthode. Ainsi, la variable `a` est du type générique qui sera précisé lors de l'appel à cette méthode. Comme `a` est un objet, nous pouvons appeler la méthode `GetType()` et la méthode `ToString()` sur cet objet.

Pour afficher un objet, nous pourrons faire :

```
int i = 5;
double d = 9.5;
string s = "abcd";
Voiture v = new Voiture();

Afficheur.Affiche<int>(i);
Afficheur.Affiche<double>(d);
Afficheur.Affiche<string>(s);
Afficheur.Affiche<Voiture>(v);
```

Dans le premier appel, nous indiquons que nous souhaitons afficher `i` dont le type générique est `int`. Tout se passe comme si le CLR créait la surcharge de la méthode `Affiche`, prenant un entier en paramètre :

```
public static void Affiche(int a)
{
    Console.WriteLine("Afficheur d'objet :");
    Console.WriteLine("\tType : " + a.GetType());
    Console.WriteLine("\tReprésentation : " + a.ToString());
}
```

De même pour l'affichage suivant, où l'on indique le type `double` entre les chevrons. C'est comme si le CLR créait la surcharge prenant un `double` en paramètre :

```
public static void Affiche(double a)
{
    Console.WriteLine("Afficheur d'objet :");
    Console.WriteLine("\tType : " + a.GetType());
    Console.WriteLine("\tReprésentation : " + a.ToString());
}
```

Et ceci pour tous les types utilisés, à savoir ici `int`, `double`, `string` et `Voiture`.

À noter que dans cet exemple, nous pouvons remplacer les quatre lignes suivantes :

```
Afficheur.Affiche<int>(i);
Afficheur.Affiche<double>(d);
Afficheur.Affiche<string>(s);
Afficheur.Affiche<Voiture>(v);
```

par :

```
Afficheur.Affiche(i);
Afficheur.Affiche(d);
Afficheur.Affiche(s);
Afficheur.Affiche(v);
```

En effet, nul besoin de préciser quel type nous souhaitons traiter ici, le compilateur est assez malin pour le déduire du type de la variable. La variable `i` étant un entier, il est obligatoire que le type générique soit un entier. Il est donc facultatif ici de le préciser.

Une fois qu'il est précisé entre les chevrons, le type générique s'utilise dans la méthode comme n'importe quel autre type. Nous pouvons avoir autant de paramètres génériques que nous le voulons dans les paramètres et utiliser le type générique dans le corps de la méthode. Par exemple, la méthode suivante :

```
public static void Echanger<T>(ref T t1, ref T t2)
{
    T temp = t1;
    t1 = t2;
    t2 = temp;
}
```

permet d'échanger le contenu de deux variables entre elles. C'est donc une méthode générique puisqu'elle précise entre les chevrons que nous pourrons utiliser le type T.

En paramètres de la méthode, nous passons deux variables de types génériques.

Dans le corps de la méthode, nous créons une variable du type générique qui sert de mémoire temporaire puis nous échangeons les références des deux variables. Nous pourrons utiliser cette méthode ainsi :

```
int i = 5;
int j = 10;
Echanger(ref i, ref j);
Console.WriteLine(i);
Console.WriteLine(j);

Voiture v1 = new Voiture { Couleur = "Rouge" };
Voiture v2 = new Voiture { Couleur = "Verte" };
Echanger(ref v1, ref v2);
Console.WriteLine(v1.Couleur);
Console.WriteLine(v2.Couleur);
```

Qui donnera :

```
10
5
Verte
Rouge
```

Il est bien sûr possible de créer des méthodes prenant en paramètres plusieurs types génériques différents. Il suffit alors de préciser autant de types différents entre les chevrons qu'il y a de types génériques différents :

```
static void Main(string[] args)
{
    int i = 5;
    int j = 5;
    double d = 9.5;

    Console.WriteLine(EstEgal(i, j));
    Console.WriteLine(EstEgal(i, d));
}
public static bool EstEgal<T, U>(T t, U u)
{
    return t.Equals(u);
}
```

Ici, la methode `EstEgal()` prend en paramètres deux types potentiellement différents. Nous l'appelons une première fois avec deux entiers et ensuite avec un entier et un double.

Créer une classe générique

Une classe générique fonctionne comme pour les méthodes. C'est une classe où nous pouvons indiquer de 1 à N types génériques. C'est comme cela que fonctionne la liste que nous avons déjà beaucoup manipulée.

En fait, la liste n'est qu'une espèce de tableau évolué. Nous pourrions très bien imaginer créer notre propre liste sur ce principe, sachant que c'est complètement absurde, car elle sera forcément moins bien que cette classe, mais c'est pour l'étude.

Le principe est d'avoir un tableau plus ou moins dynamique qui grossit si jamais le nombre d'éléments devient trop grand pour sa capacité.

Pour déclarer une classe générique, nous utiliserons à nouveau les chevrons à la fin de la ligne qui déclare la classe :

```
public class MaListeGenerique<T>
{
}
```

Nous allons réaliser une implémentation toute basique de cette classe histoire de voir un peu à quoi ressemble une classe générique. Cette classe n'a d'intérêt que pour étudier les génériques, vous lui préférerez évidemment la classe `List<>` du framework .NET.

Nous avons besoin de trois variables privées. La capacité de la liste, le nombre d'éléments dans la liste et le tableau générique.

```
public class MaListeGenerique<T>
{
    private int capacite;
    private int nbElements;
    private T[] tableau;

    public MaListeGenerique()
    {
        capacite = 10;
        nbElements = 0;
        tableau = new T[capacite];
    }
}
```

Remarquez la déclaration du tableau. Elle utilise le type générique. Concrètement, cela veut dire que quand nous utiliserons la liste avec un entier, nous aurons un tableau d'entiers. Lorsque nous utiliserons la liste avec un objet `Voiture`, nous aurons un tableau de `Voiture`, etc.

Nous initialisons ces variables membres dans le constructeur, en décidant complètement arbitrairement que la capacité par défaut de notre liste est de 10 éléments. La dernière ligne instancie le tableau en lui indiquant sa taille.

Il reste à implémenter l'ajout dans la liste :

```
public class MaListeGenerique<T>
```

```
{
    [Code enlevé pour plus de clarté]

    public void Ajouter(T element)
    {
        if (nbElements >= capacite)
        {
            capacite *= 2;
            T[] copieTableau = new T[capacite];
            for (int i = 0; i < tableau.Length; i++)
            {
                copieTableau[i] = tableau[i];
            }
            tableau = copieTableau;
        }
        tableau[nbElements] = element;
        nbElements++;
    }
}
```

Il s'agit simplement de mettre la valeur que l'on souhaite ajouter à l'emplacement adéquat dans le tableau. Nous le mettons en dernière position, c'est-à-dire à l'emplacement correspondant au nombre d'éléments.

Au début, nous avons commencé par vérifier si le nombre d'éléments était supérieur à la capacité du tableau. Si c'est le cas, alors nous devons augmenter la capacité du tableau. J'ai ici décidé encore complètement arbitrairement que je doublais la capacité. Il ne reste plus qu'à créer un nouveau tableau de cette nouvelle capacité et à copier les éléments du premier tableau dans celui-ci.

Vous aurez noté que le paramètre de la méthode `Ajouter` est bien du type générique.

Pour le plaisir, rajoutons enfin une méthode permettant de récupérer un élément d'indice donné :

```
public class MaListeGenerique<T>
{
    [Code enlevé pour plus de clarté]

    public T ObtenirElement(int indice)
    {
        return tableau[indice];
    }
}
```

Il s'agit juste d'accéder au tableau pour renvoyer la valeur à l'indice concerné. L'élément intéressant ici est de constater que le type de retour de la méthode est bien du type générique.

Cette liste peut s'utiliser de la manière suivante :

```
MaListeGenerique<int> maListe = new MaListeGenerique<int>();
maListe.Ajouter(25);
```

```
maListe.Ajouter(30);
maListe.Ajouter(5);

Console.WriteLine(maListe.ObtenirElement(0));
Console.WriteLine(maListe.ObtenirElement(1));
Console.WriteLine(maListe.ObtenirElement(2));

for (int i = 0; i < 30; i++)
{
    maListe.Ajouter(i);
}
```

Ici, nous utilisons la liste avec un entier, mais elle fonctionnerait tout aussi bien avec un autre type. N'hésitez pas à passer en debug dans la méthode `Ajouter()` pour observer ce qui se passe exactement lors de l'augmentation de capacité.

Voilà comment on crée une classe générique !

Une fois qu'on a compris que le type générique s'utilise comme n'importe quel autre type, cela devient assez facile.



Rappelez-vous, toute classe qui manipule des `object` est susceptible d'être améliorée en utilisant les génériques.

La valeur par défaut d'un type générique

Vous aurez remarqué dans l'implémentation de la liste du dessus que si nous essayons d'obtenir un élément du tableau à un indice qui n'existe pas, nous aurons une erreur. Ce comportement est une bonne chose, car il est important de gérer les cas aux limites. En l'occurrence ici, on délègue au tableau la gestion du cas limite.

On pourrait envisager de gérer nous-mêmes ce cas limite en affichant un message et en renvoyant une valeur nulle, mais ceci pose un problème. Pour un objet `Voiture`, qui est un type référence, il est tout à fait pertinent d'avoir `null` ; pour un `int`, qui est un type valeur, ça n'a pas de sens. C'est là qu'intervient le mot-clé `default`. Comme son nom l'indique, il renvoie la valeur par défaut du type. Pour un type référence, c'est `null` ; pour un type valeur c'est 0. Ce qui donnerait :

```
public T ObtenirElement(int indice)
{
    if (indice < 0 || indice >= nbElements)
    {
        Console.WriteLine("L'indice n'est pas bon");
        return default(T);
    }
    return tableau[indice];
}
```

Les interfaces génériques

Les interfaces peuvent aussi être génériques. D'ailleurs, ça me fait penser que plus haut, je vous ai indiqué qu'un certain code n'était pas très esthétique et que j'en parlerai plus tard... Le moment est venu ! Vous ne vous en souvenez plus ? Petit rappel pour les étourdis : il s'agissait du chapitre sur les interfaces où nous avions implémenté l'interface `IComparable`.

Nous souhaitions comparer des voitures entre elles et nous avions obtenu le code suivant :

```
public class Voiture : IComparable
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }

    public int CompareTo(object obj)
    {
        Voiture voiture = (Voiture)obj;
        return Vitesse.CompareTo(voiture.Vitesse);
    }
}
```

Je souhaite pouvoir comparer des voitures entre elles, mais le framework .NET me fournit un `object` en paramètres de la méthode `CompareTo()`. Quelle idée ! Comme si je voulais comparer des voitures avec des chats ou des chiens. Cela me force en plus à faire un cast. Pourquoi il ne me passe pas directement une `Voiture` en paramètre ?

Vous en avez l'intuition ? Un `object` ! Oui, mais c'est un peu lourd à manier... et je ne parle pas des performances !

C'est là où les génériques vont voler à notre secours. L'interface `IComparable` date de la première version du framework .NET. Le C# ne possédait pas encore les types génériques. Depuis leur apparition, il est possible d'implémenter la version générique de cette interface.

Pour cela, nous faisons suivre l'interface du type que nous souhaitons utiliser entre les chevrons. Cela donne :

```
public class Voiture : IComparable<Voiture>
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }

    public int CompareTo(Voiture obj)
    {
        return Vitesse.CompareTo(obj.Vitesse);
    }
}
```

Nous devons toujours implémenter la méthode `CompareTo()` sauf que nous avons désormais un objet `Voiture` en paramètre, ce qui nous évite de le caster.

Les restrictions sur les types génériques

Une méthode ou une classe générique c'est bien, mais peut-être voulons-nous qu'elles ne fonctionnent pas avec tous les types. Aussi, le C# permet de définir des restrictions sur les types génériques. Pour ce faire, on utilise le mot-clé `where`.

Il existe six types de restrictions :

Contrainte	Description
`where` T : `struct`	Le type générique doit être un type valeur
`where` T : `class`	Le type générique doit être un type référence
`where` T : `new()`	Le type générique doit posséder un constructeur par défaut
`where` T : `IMonInterface`	Le type générique doit implémenter l'interface `IMonInterface`
`where` T : `MaClasse`	Le type générique doit dériver de la classe `MaClasse`
`where` T1 : `T2`	Le type générique doit dériver du type générique `T2`

Par exemple, nous pouvons définir une restriction sur une méthode générique afin qu'elle n'accepte en type générique que des types qui implémentent une interface.

Soit l'interface suivante :

```
public interface IVolant
{
    void DeplierLesAiles();
    void Voler();
}
```

qui est implémentée par deux objets :

```
public class Avion : IVolant
{
    public void DeplierLesAiles()
    {
        Console.WriteLine("Je déplie mes ailes mécaniques");
    }

    public void Voler()
    {
        Console.WriteLine("J'allume le moteur");
    }
}

public class Oiseau : IVolant
{
```

```
    public void DeplierLesAiles()
    {
        Console.WriteLine("Je déplie mes ailes d'oiseau");
    }

    public void Voler()
    {
        Console.WriteLine("Je bas des ailes");
    }
}
```

Nous pouvons créer une méthode générique qui s'occupe d'instancier ces objets et d'appeler les méthodes de l'interface :

```
public static T Creer<T>() where T : IVolant, new()
{
    T t = new T();
    t.DeplierLesAiles();
    t.Voler();
    return t;
}
```

Ici, la restriction porte sur deux niveaux. Il faut dans un premier temps que le type générique implémente l'interface `IVolant`. En outre, il faut qu'il possède un constructeur, bref qu'il soit instanciable.

Nous pouvons donc utiliser cette méthode de cette façon :

```
Oiseau oiseau = Creer<Oiseau>();
Avion a380 = Creer<Avion>();
```

Nous appelons la méthode `Créer()` avec le type générique `Oiseau`, qui implémente bien `IVolant` et qui est aussi instanciable. Grâce à cela, nous pouvons utiliser l'opérateur `new` pour créer notre type générique, appeler les méthodes de l'interface et renvoyer l'instance. Ce qui donne :

```
Je déplie mes ailes d'oiseau
Je bas des ailes
Je déplie mes ailes mécaniques
J'allume le moteur
```

Si nous tentons d'utiliser la méthode avec un type qui n'implémente pas l'interface `IVolant`, comme :

```
Voiture v = Creer<Voiture>();
```

Nous aurons l'erreur de compilation suivante :

```
Le type 'MaPremiereApplication.Voiture' ne peut pas être utilisé
     comme paramètre de type 'T' dans le type ou la méthode géné
    rique 'MaPremiereApplication.Program.Creer<T>()'. Il n'y a
```

```
    pas de conversion de référence implicite de '
    MaPremiereApplication.Voiture' en 'MaPremiereApplication.
    IVolant'.
```

Globalement, il nous dit que l'objet `Voiture` n'implémente pas `IVolant`.



Oui, mais dans ce cas, plutôt que d'utiliser une méthode générique, pourquoi la méthode ne renvoie pas `IVolant` ?

C'est une judicieuse remarque, mais elle implique quelques modifications de code. En effet, il faudrait indiquer quel type instancier.

Nous pourrions par exemple faire :

```
public enum TypeDeVolant
{
    Oiseau,
    Avion
}

public static IVolant Creer(TypeDeVolant type)
{
    IVolant volant;
    switch (type)
    {
        case TypeDeVolant.Oiseau:
            volant = new Oiseau();
            break;
        case TypeDeVolant.Avion:
            volant = new Avion();
            break;
        default:
            return null;
    }
    volant.DeplierLesAiles();
    volant.Voler();
    return volant;
}
```

Et instancier nos objets de cette façon :

```
Oiseau oiseau = (Oiseau)Creer(TypeDeVolant.Oiseau);
Avion a380 = (Avion)Creer(TypeDeVolant.Avion);
```

Ce qui complique un peu les choses et rajoute des casts dont on pourrait volontiers se passer. De plus, si nous créons un nouvel objet qui implémente cette interface, il faudrait tout modifier.

Avouez qu'avec les types génériques, c'est quand même plus propre ! Nous pouvons bien sûr avoir des restrictions sur les types génériques d'une classe.

Pour le montrer, nous allons créer une classe dont l'objectif est de disposer de types valeur qui pourraient ne pas avoir de valeur. Pour les types référence, il suffit d'utiliser le mot-clé `null`. Mais pour les types valeur comme les entiers, nous n'avons rien pour indiquer que ceux-ci n'ont pas de valeur.

Par exemple :

```
public class TypeValeurNull<T> where T : struct
{
    private bool aUneValeur;
    public bool AUneValeur
    {
        get { return aUneValeur; }
    }

    private T valeur;
    public T Valeur
    {
        get
        {
            if (aUneValeur)
                return valeur;
            throw new InvalidOperationException();
        }
        set
        {
            aUneValeur = true;
            valeur = value;
        }
    }
}
```

Ici, nous utilisons une classe possédant un type générique qui sera un type valeur, grâce à la condition `where T : struct`. Cette classe encapsule le type générique pour indiquer avec un booléen si le type a une valeur ou pas. Ne faites pas attention à la ligne :

```
throw new InvalidOperationException();
```

qui permet juste de renvoyer une erreur, nous étudierons les exceptions un peu plus loin. Elle pourra s'utiliser ainsi :

```
TypeValeurNull<int> entier = new TypeValeurNull<int>();
if (!entier.AUneValeur)
{
    Console.WriteLine("l'entier n'a pas de valeur");
}
entier.Valeur = 5;
if (entier.AUneValeur)
{
    Console.WriteLine("Valeur de l'entier : " + entier.Valeur);
}
```

Et si nous souhaitons faire de même pour un autre type valeur, il n'y a rien à coder de plus :

```
TypeValeurNull<double> valeur = new TypeValeurNull<double>();
```

C'est quand même super pratique comme classe ! ! Mais ne rêvons pas, cette idée ne vient pas de moi. C'est en fait une fonctionnalité du framework .NET : les types nullables.

Les types nullables

En fait, la classe que nous avons vue au-dessus existe déjà dans le framework .NET, et en mieux ! Évidemment. Elle fait exactement ce que j'ai décrit, c'est-à-dire permettre à un type valeur d'avoir une valeur nulle.

Elle est mieux faite dans la mesure où elle tire parti de certaines fonctionnalités du framework .NET qui en simplifient l'écriture. Il s'agit de la classe `Nullable<>`.

Aussi, nous pourrons créer un entier pouvant être `null` grâce au code suivant :

```
Nullable<int> entier = null;
if (!entier.HasValue)
{
    Console.WriteLine("l'entier n'a pas de valeur");
}
entier = 5;
if (entier.HasValue)
{
    Console.WriteLine("Valeur de l'entier : " + entier.Value);
}
```

Le principe est grosso modo le même sauf que nous pouvons utiliser le mot-clé `null` ou affecter directement la valeur à l'entier en utilisant l'opérateur d'affectation, sans passer par la propriété `Valeur`. Il peut aussi être comparé au mot-clé `null` ou être utilisé avec l'opérateur +, etc. Ceci est possible grâce à certaines fonctionnalités du C# que nous n'avons pas vues et qui sortent de l'étude de ce livre.

Cette classe est tellement pratique que le compilateur a été optimisé pour simplifier son écriture. En effet, utiliser `Nullable<>` est un peu long pour nous autres informaticiens qui sommes des paresseux !

Aussi, l'écriture :

```
Nullable<int> entier = null;
```

peut se simplifier en :

```
int? entier = null;
```

C'est le point d'interrogation qui remplace la déclaration de la classe `Nullable<>`.

En résumé

- Avec les génériques, vous pouvez créer des méthodes ou des classes qui sont indépendantes d'un type. On les appellera des méthodes génériques et des types génériques.
- On utilise les chevrons <> pour indiquer le type d'une classe ou d'une méthode générique.
- Les interfaces peuvent aussi être génériques, comme l'interface `IEnumerable<>`.
- Les types nullables constituent un exemple d'utilisation très pratique des classes génériques.

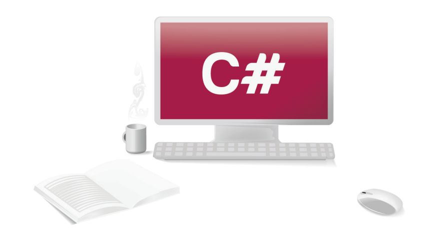
Chapitre 28

TP : types génériques

Difficulté : 

Ah , un peu de pratique histoire de vérifier que nous avons bien compris les génériques. C'est un concept assez facile à appréhender mais relativement difficile à mettre en œuvre. Quand en ai-je besoin ? Comment ?

Voici donc un petit exercice qui va vous permettre d'essayer de mettre en œuvre une classe générique.



Instructions pour réaliser la première partie du TP

Cet exercice va se dérouler en deux parties. Dans la première partie du TP, nous allons réaliser une liste chaînée. Il s'agit du grand classique des TP d'informatique en C. Je vous en rappelle le principe.

La liste chaînée permet de naviguer d'élément en élément. Quand nous sommes sur le premier élément, le suivant est accessible par sa propriété `Suivant`. Lorsque nous accédons au suivant, l'élément précédent est accessible par la propriété `Precedent` et le suivant toujours accessible par la propriété `Suivant`. S'il n'y a pas de précédent ou pas de suivant, l'élément est `null` (voir figure 28.1).

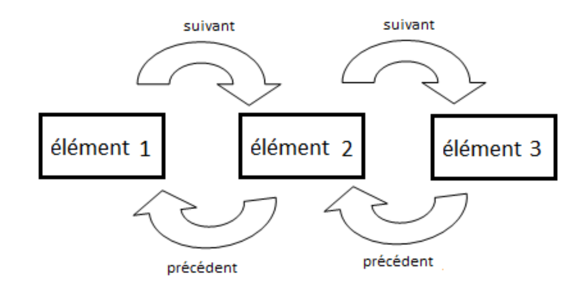


FIGURE 28.1 – Description de la liste chaînée

Si on insère un élément à la position 1, les autres se décalent, ainsi qu'illustré à la figure 28.2.

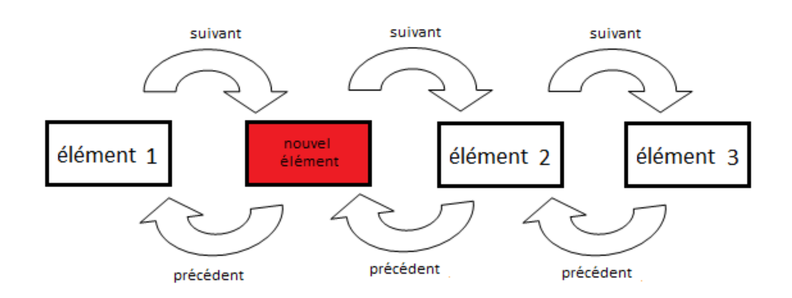


FIGURE 28.2 – Insertion d'un nouvel élément dans la liste chaînée

Voilà, il faut donc créer une telle liste chaînée d'éléments. Le but est bien sûr de faire en sorte que l'élément soit générique.

N'hésitez pas à réfléchir un peu avant de vous lancer. Cela pourrait paraître un peu simpliste, mais en fait cela occasionne quelques nœuds au cerveau.

Toujours est-il que je souhaiterais disposer d'une propriété en lecture seule permettant d'accéder au premier élément ainsi qu'une autre propriété également en lecture seule

permettant d'accéder au dernier élément. Bien sûr, il faut pouvoir naviguer d'élément en élément avec des propriétés `Precedent` et `Suivant`.

Il faut évidemment une méthode permettant d'ajouter un élément à la fin de la liste. Nous aurons également besoin d'une méthode permettant d'accéder à un élément à partir de son indice et enfin d'une méthode permettant d'insérer un élément à un indice, décalant tous les suivants.

Voilà pour la création de la classe.

Ensuite, notre programme instanciera notre liste chaînée pour lui ajouter les entiers 5, 10 et 4. Puis nous afficherons les valeurs de cette liste en nous basant sur la première propriété et en naviguant d'élément en élément.

Nous afficherons ensuite les différents éléments en utilisant la méthode d'accès à un élément par son indice.

Enfin, nous insérerons la valeur 99 à la première position (position 0), puis la valeur 33 à la deuxième position et enfin la valeur 30 à nouveau à la deuxième position.

Puis nous afficherons tout ce beau monde.

Fin de l'énoncé, ouf !

Pour ceux qui n'ont pas besoin d'aide, les explications sont terminées. Ouvrez vos Visual C# Express (ou vos Visual Studio si vous êtes riches !) et à vos claviers.

Pour les autres, je vais essayer de vous guider un peu plus en essayant tout de même de ne pas trop vous donner d'indications non plus.

En fait, votre liste chaînée n'est pas vraiment une liste, comme pourrait l'être la `List<>` que nous connaissons. Cette liste chaînée possède un point d'entrée qui est le premier élément. L'ajout du premier élément est très simple, il suffit de mettre à jour une propriété. Pour ajouter l'élément suivant, il faut en fait brancher la propriété `Suivant` du premier élément à l'élément que nous sommes en train d'ajouter. Et inversement, la propriété `Precedent` de l'élément que nous souhaitons ajouter sera mise à jour avec le premier élément.

On se rend compte que l'élément est un peu plus complexe qu'un simple type. Nous allons donc avoir une classe générique possédant trois propriétés (`Precedent`, `Suivant` et `Valeur`). Et nous aurons également une classe du même type générique possédant la propriété `Premier` et la propriété `Dernier` et les méthodes d'ajout, d'obtention de l'élément et d'insertion.

Allez, je vous en ai assez dit. À vous de jouer !

Correction

Pas si facile hein ? Mais bon, comme vous êtes super entraînés, cela n'a pas dû vous poser trop de problèmes.

Voici la correction que je propose.

La première chose à faire est de créer la classe générique permettant de stocker un

élément :

```
public class Chainage<T>
{
    public Chainage<T> Precedent { get; set; }
    public Chainage<T> Suivant { get; set; }
    public T Valeur { get; set; }
}
```

C'est une classe générique toute simple qui possède une valeur du type générique et deux propriétés du même type que l'élément pour obtenir le précédent ou le suivant.

Peut-être que la plus grande difficulté réside, ici, dans le fait de bien modéliser la classe qui permet d'encapsuler l'élément.

Il faudra ensuite créer la liste générique et ses méthodes :

```
public class ListeChainee<T>
{
}
```

La liste chaînée possède également un type générique. Nous créons sa propriété `Premier` :

```
public class ListeChainee<T>
{
    public Chainage<T> Premier { get; private set; }
}
```

Là, c'est très simple, il s'agit juste d'une propriété en lecture seule stockant le premier élément. C'est la méthode `Ajouter()` qui permettra de mettre à jour cette valeur. Notez quand même que nous utilisons le type générique comme type générique de la classe encapsulante.

Par contre, pour la propriété `Dernier`, c'est un peu plus compliqué. Pour la retrouver, nous allons parcourir tous les éléments à partir de la propriété `Premier`. Ce qui donne :

```
public class ListeChainee<T>
{
    [...Code supprimé pour plus de clarté...]

    public Chainage<T> Dernier
    {
        get
        {
            if (Premier == null)
                return null;
            Chainage<T> dernier = Premier;
            while (dernier.Suivant != null)
            {
                dernier = dernier.Suivant;
            }
            return dernier;
        }
```

```
    }
}
```

On parcourt les éléments en bouclant sur la propriété `Suivant`, tant que celle-ci n'est pas nulle. Il s'agit là d'un parcours assez classique où on utilise une variable temporaire qui passe au suivant à chaque itération.

Nous pouvons à présent créer la méthode `Ajouter` :

```
public class ListeChainee<T>
{
    [...Code supprimé pour plus de clarté...]

    public void Ajouter(T element)
    {
        if (Premier == null)
        {
            Premier = new Chainage<T> { Valeur = element };
        }
        else
        {
            Chainage<T> dernier = Dernier;
            dernier.Suivant = new Chainage<T> { Valeur =
                element, Precedent = dernier };
        }
    }
}
```

Cette méthode traite tout d'abord le cas du premier élément. Il s'agit simplement de mettre à jour la propriété `Premier`. De même, grâce au calcul interne de la propriété `Dernier`, il sera facile d'ajouter un nouvel élément en se branchant sur la propriété `Suivant` du dernier élément.

Notez que vu que nous ne la renseignons pas, la propriété `Suivant` du nouvel élément sera bien à `null`.

Pour obtenir un élément à un indice donné, il suffira de reprendre le même principe que lors du parcours pour obtenir le dernier élément, sauf qu'il faudra s'arrêter au bon moment :

```
public class ListeChainee<T>
{
    [...Code supprimé pour plus de clarté...]

    public Chainage<T> ObtenirElement(int indice)
    {
        Chainage<T> temp = Premier;
        for (int i = 1; i <= indice; i++)
        {
            if (temp == null)
                return null;
            temp = temp.Suivant;
```

```
        }
        return temp;
    }
}
```

Ici, plusieurs solutions. J'ai choisi d'utiliser une boucle `for`. Nous aurions très bien pu garder la boucle `while` comme pour la propriété `Dernier`.

Enfin, il ne reste plus qu'à insérer un élément :

```
public class ListeChainee<T>
{
    [...Code supprimé pour plus de clarté...]

    public void Inserer(T element, int indice)
    {
        if (indice == 0)
        {
            Chainage<T> temp = Premier;
            Premier = new Chainage<T> { Suivant = temp, Valeur = element };
            temp.Precedent = Premier;
        }
        else
        {
            Chainage<T> elementAIndice = ObtenirElement(indice);
            if (elementAIndice == null)
                Ajouter(element);
            else
            {
                Chainage<T> precedent = elementAIndice.Precedent;
                Chainage<T> temp = precedent.Suivant;
                precedent.Suivant = new Chainage<T> { Valeur = element, Precedent = precedent, Suivant = temp };
            }
        }
    }
}
```

Nous traitons, dans un premier temps, le cas où l'on doit insérer en tête. Il suffit de mettre à jour la valeur du premier en ayant au préalable décalé ce dernier d'un cran. Attention, si `Premier` est `null`, nous allons avoir un problème. Dans ce cas, soit nous laissons le problème ; en effet, peut-on vraiment insérer un élément avant les autres s'il n'y en a pas ? Soit nous gérons le cas et décidons d'insérer l'élément en tant que `Premier` :

```
public class ListeChainee<T>
{
```

```
    [...Code supprimé pour plus de clarté...]

    public void Inserer(T element, int indice)
    {
        if (indice == 0)
        {
            if (Premier == null)
                Premier = new Chainage<T> { Valeur = element };
            else
            {
                Chainage<T> temp = Premier;
                Premier = new Chainage<T> { Suivant = temp,
                    Valeur = element };
                temp.Precedent = Premier;
            }
        }
        else
        {
            Chainage<T> elementAIndice = ObtenirElement(indice)
                ;
            if (elementAIndice == null)
                Ajouter(element);
            else
            {
                Chainage<T> precedent = elementAIndice.
                    Precedent;
                Chainage<T> temp = precedent.Suivant;
                precedent.Suivant = new Chainage<T> { Valeur =
                    element, Precedent = precedent, Suivant =
                    temp };
            }
        }
    }
}
```

Pour les autres cas, si nous tentons d'insérer à un indice qui n'existe pas, nous insérons à la fin en utilisant la méthode `Ajouter()` existante. Sinon, on intercale le nouvel élément dans la liste en prenant soin de brancher le précédent sur notre nouvel élément et de brancher le suivant sur notre nouvel élément.

Voilà pour notre classe. Il reste à l'utiliser :

```
static void Main(string[] args)
{
    ListeChainee<int> listeChainee = new ListeChainee<int>();
    listeChainee.Ajouter(5);
    listeChainee.Ajouter(10);
    listeChainee.Ajouter(4);
    Console.WriteLine(listeChainee.Premier.Valeur);
    Console.WriteLine(listeChainee.Premier.Suivant.Valeur);
    Console.WriteLine(listeChainee.Premier.Suivant.Suivant.
```

```
        Valeur);
    Console.WriteLine("*************");
    Console.WriteLine(listeChainee.ObtenirElement(0).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(1).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(2).Valeur);
    Console.WriteLine("*************");
    listeChainee.Inserer(99, 0);
    listeChainee.Inserer(33, 2);
    listeChainee.Inserer(30, 2);
    Console.WriteLine(listeChainee.ObtenirElement(0).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(1).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(2).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(3).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(4).Valeur);
    Console.WriteLine(listeChainee.ObtenirElement(5).Valeur);
}
```

Ce qui affichera donc :

```
5
10
4
*************
5
10
4
*************
99
5
30
33
10
4
```

Instructions pour réaliser la deuxième partie du TP

Bon, c'est très bien de pouvoir accéder à un élément par son indice. Mais une liste sur laquelle on ne peut pas faire un `foreach`, c'est quand même bien dommage.

Attaquons désormais la deuxième partie du TP. Toujours dans l'optique de manipuler les génériques, nous allons faire en sorte que notre liste chaînée puisse être parcourue en utilisant un `foreach`.

Nous avons dit plus haut qu'il suffisait d'implémenter l'interface `IEnumerable`. En l'occurrence, nous allons implémenter sa version générique, vu que nous travaillons avec une classe générique.

Voilà le but de ce TP !

Si vous vous le sentez, allez-y ! Je pense quand même que vous allez avoir besoin d'être un peu guidés car c'est une opération assez particulière.

Vous l'aurez deviné, il faut que notre liste implémente l'interface `IEnumerable<T>`.

Le fait d'implémenter cette interface va vous forcer à implémenter deux méthodes `GetEnumerator()`, la version normale et la version explicite. Sachez dès à présent que les deux méthodes feront exactement la même chose.



Mais, qu'est-ce qu'il raconte ? Implémenter une interface explicitement ? On n'a jamais vu ça !

C'est vrai ! Allez, je vous en parle après la correction. Pour l'instant, cela ne devrait pas vous perturber, car les deux méthodes font exactement la même chose.

En l'occurrence, elles renverront un `Enumerator` personnalisé.

Il va donc falloir créer cet énumérateur qui va s'occuper de la mécanique permettant de naviguer dans notre liste. Il s'agit d'une nouvelle classe qui va devoir implémenter l'interface `IEnumerator<T>`, c'est-à-dire :

```
public class ListeChaineeEnumerator<T> : IEnumerator<T>
{
}
```

Cette interface permet d'indiquer que notre énumérateur va respecter le contrat lui permettant de fonctionner avec un `foreach`. Avec cette interface, vous allez devoir implémenter :

– la propriété `Current` ;
– la propriété explicite `Current` (qui sera la même chose que la précédente) ;
– la méthode `MoveNext` qui permet de passer à l'élément suivant ;
– la méthode `Reset`, qui permet de revenir au début de la liste ;
– la méthode `Dispose`.

La méthode `Dispose` est en fait héritée de l'interface `IDisposable` dont hérite l'interface `IEnumerator<T>`. C'est une interface particulière qui offre l'opportunité de faire tout ce qu'il faut pour nettoyer la classe, c'est-à-dire libérer les variables qui en auraient besoin. En l'occurrence, ici nous n'aurons rien à faire mais il faut quand même que la méthode soit présente. Elle sera donc vide.

Pour implémenter les autres méthodes, il faut que l'énumérateur connaisse la liste qu'il doit énumérer. Il faudra donc que la classe `ListeChaineeEnumerator` prenne en paramètre de son constructeur la liste à énumérer. Dans ce constructeur, on initialise la variable membre indice qui contient l'indice courant. La propriété `Current` renverra l'élément à l'indice courant. La méthode `MoveNext` passe à l'élément suivant et renvoie faux s'il n'y a plus d'éléments et vrai sinon. Enfin la méthode `Reset` repasse l'indice à sa valeur initiale.

À noter que la valeur initiale de l'indice est `-1`, car la boucle `foreach` commence par appeler la méthode `MoveNext` qui commence par aller à l'élément suivant, c'est-à-dire

à l'élément 0.

Il ne reste plus qu'à vous dire exactement quoi mettre dans les méthodes `GetEnumerator` de la liste chaînée, car vous ne trouverez peut-être pas du premier coup :

```
public IEnumerator<T> GetEnumerator()
{
    return new ListeChaineeEnumerator<T>(this);
}

IEnumerator IEnumerable.GetEnumerator()
{
    return new ListeChaineeEnumerator<T>(this);
}
```

C'est à vous de jouer pour la suite.

Correction

Encore moins facile ! Tant qu'on ne l'a pas fait une première fois, implémenter l'interface `IEnumerable` est un peu déroutant. Après, c'est toujours pareil.

Voici donc ma correction.

Tout d'abord, la liste chaînée doit implémenter `IEnumerable<T>`, ce qui donne :

```
public class ListeChainee<T> : IEnumerable<T>
{
    [...Code identique au TP précédent...]

    public IEnumerator<T> GetEnumerator()
    {
        return new ListeChaineeEnumerator<T>(this);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new ListeChaineeEnumerator<T>(this);
    }
}
```

Là, c'est du tout cuit vu que je vous avais donné la solution un peu plus tôt . J'espère que vous avez au moins réussi ça !

Maintenant, il faut donc créer un nouvel énumérateur personnalisé en lui passant notre liste chaînée en paramètre. Cet énumérateur doit implémenter l'interface `IEnumerator`, ce qui donne :

```
public class ListeChaineeEnumerator<T> : IEnumerator<T>
{
}
```

Comme prévu, il faut donc un constructeur qui prend en paramètre la liste chaînée :

```
public class ListeChaineeEnumerator<T> : IEnumerator<T>
{
    private int indice;
    private ListeChainee<T> listeChainee;
    public ListeChaineeEnumerator(ListeChainee<T> liste)
    {
        indice = -1;
        listeChainee = liste;
    }

    public void Dispose()
    {
    }
}
```

Cette liste sera enregistrée dans une variable membre de la classe. Tant que nous y sommes, nous ajoutons un indice privé que nous initialisons à -1, comme déjà expliqué.

Notez également que la méthode `Dispose()` est vide. Il reste à implémenter les autres méthodes :

```
public class ListeChaineeEnumerator<T> : IEnumerator<T>
{
    private int indice;
    private ListeChainee<T> listeChainee;
    public ListeChaineeEnumerator(ListeChainee<T> liste)
    {
        indice = -1;
        listeChainee = liste;
    }

    public void Dispose()
    {
    }

    public bool MoveNext()
    {
        indice++;
        Chainage<T> element = listeChainee.ObtenirElement(indice);
        return element != null;
    }

    public T Current
    {
        get
        {
            Chainage<T> element = listeChainee.ObtenirElement(indice);
            if (element == null)
```

```
                    return default(T);
                return element.Valeur;
            }
        }

        object IEnumerator.Current
        {
            get { return Current; }
        }

        public void Reset()
        {
            indice = -1;
        }
}
```

Commençons par la méthode `MoveNext()`. Elle passe à l'indice suivant et renvoie faux ou vrai, selon qu'on arrive au bout de la liste ou pas. N'oubliez pas que c'est la première méthode qui sera appelée dans le `foreach`, donc pour passer à l'élément suivant, on incrémente l'indice pour le positionner à l'élément 0. C'est pour cela que l'indice a été initialisé à -1. On utilise ensuite la méthode existante de la liste pour obtenir l'élément à un indice afin de savoir si notre liste peut continuer à s'énumérer.

La propriété `Current` renvoie l'élément à l'indice courant, pour cela on utilise l'indice pour accéder à l'élément courant, en utilisant les méthodes de la liste. L'autre propriété `Current` fait la même chose, il suffit de l'appeler.

Enfin, la méthode `Reset` permet de réinitialiser l'énumérateur en retournant à l'indice initial.

Finalement, ce n'est pas si compliqué que ça. Mais il faut avouer que la première fois, c'est un peu déroutant !

À mon sens, c'est un bon exercice pratique. Peut-être que mes explications ont suffi à vous guider. Sans doute avez-vous dû regarder un peu la documentation de `IEnumerable` sur internet. Peut-être avez-vous cherché des ressources traitant du même sujet. Dans tous les cas, devoir implémenter une interface du framework .NET est une situation que vous allez fréquemment devoir rencontrer. Il est bon de s'y entraîner.

Aller plus loin

Vous me direz qu'il fallait le deviner, qu'on avait besoin d'une classe indépendante qui permettait de gérer l'énumérateur !

En fait, ce n'est pas obligatoire. On peut très bien faire en sorte que notre classe gère la liste chaînée et son énumérateur. Il suffit de faire en sorte que la liste chaînée implémente également `IEnumerator<T>` et de gérer la logique à l'intérieur de la classe.

Par contre, ce n'est pas recommandé. D'une manière générale il est bon qu'une classe n'ait à s'occuper que d'une seule chose. On appelle ça le principe de **responsabilité**

unique[1]. Plus une classe fait de choses, plus une modification impacte les autres choses. Ici, il est judicieux de garder le découplage des deux classes.

Il y a quand même un élément que l'on peut améliorer dans le code de la correction. En effet, cette liste n'est pas extrêmement optimisée car lorsque nous obtenons un élément, nous reparcourons toute la liste depuis le début, notamment dans le cas de la gestion de l'énumérateur. Il pourrait être judicieux qu'à chaque `foreach`, nous ne parcourions pas tous les éléments et qu'on évite d'appeler continuellement la méthode `ObtenirElement()`. Une idée ? Cela pourrait se faire en éliminant l'indice et en utilisant une variable de type `Chainage<T>`, par exemple :

```
public class ListeChaineeEnumerator<T> : IEnumerator<T>
{
    private Chainage<T> courant;
    private ListeChainee<T> listeChainee;
    public ListeChaineeEnumerator(ListeChainee<T> liste)
    {
        courant = null;
        listeChainee = liste;
    }

    public void Dispose()
    {
    }

    public bool MoveNext()
    {
        if (courant == null)
            courant = listeChainee.Premier;
        else
            courant = courant.Suivant;

        return courant != null;
    }

    public T Current
    {
        get
        {
            if (courant == null)
                return default(T);
            return courant.Valeur;
        }
    }

    object IEnumerator.Current
    {
        get { return Current; }
    }
```

1. En anglais, SRP, pour *Single Responsibility Principle*.

```

    public void Reset()
    {
        courant = null;
    }
}
```

Ici, c'est la variable `courant` qui nous permet d'itérer au fur et à mesure de la liste chaînée. C'est le même principe que dans la méthode `ObtenirElement`, sauf qu'on ne reparcourt pas toute la liste à chaque fois.

Ici, cette optimisation est négligeable pour notre utilisation. Elle peut s'avérer intéressante si la liste grossit énormément.

Dans tous les cas, ça ne fait pas de mal d'aller plus vite !

Remarquons avant de terminer qu'il est possible de simplifier encore la classe grâce à un mot-clé que nous découvrirons dans la partie suivante : `yield`. Il permet de créer facilement des énumérateurs. Ce qui fait que le code complet de la liste chaînée pourra être :

```
public class ListeChainee<T> : IEnumerable<T>
{
    public Chainage<T> Premier { get; private set; }

    public Chainage<T> Dernier
    {
        get
        {
            if (Premier == null)
                return null;
            Chainage<T> dernier = Premier;
            while (dernier.Suivant != null)
            {
                dernier = dernier.Suivant;
            }
            return dernier;
        }
    }

    public void Ajouter(T element)
    {
        if (Premier == null)
        {
            Premier = new Chainage<T> { Valeur = element };
        }
        else
        {
            Chainage<T> dernier = Dernier;
            dernier.Suivant = new Chainage<T> { Valeur =
                element, Precedent = dernier };
        }
```

```
    }

    public Chainage<T> ObtenirElement(int indice)
    {
        Chainage<T> temp = Premier;
        for (int i = 1; i <= indice; i++)
        {
            if (temp == null)
                return null;
            temp = temp.Suivant;
        }
        return temp;
    }

    public void Inserer(T element, int indice)
    {
        if (indice == 0)
        {
            if (Premier == null)
                Premier = new Chainage<T> { Valeur = element };
            else
            {
                Chainage<T> temp = Premier;
                Premier = new Chainage<T> { Suivant = temp,
                    Valeur = element };
                temp.Precedent = Premier;
            }
        }
        else
        {
            Chainage<T> elementAIndice = ObtenirElement(indice)
                ;
            if (elementAIndice == null)
                Ajouter(element);
            else
            {
                Chainage<T> precedent = elementAIndice.
                    Precedent;
                Chainage<T> temp = precedent.Suivant;
                precedent.Suivant = new Chainage<T> { Valeur =
                    element, Precedent = precedent, Suivant =
                    temp };
            }
        }
    }

    public IEnumerator<T> GetEnumerator()
    {
        Chainage<T> courant = Premier;
        while (courant != null)
```

```
            {
                yield return courant.Valeur;
                courant = courant.Suivant;
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
}
```

Remarquons que nous n'avons plus besoin de la classe `ListeChaineeEnumerator`. L'implémentation devient très facile. Nous reviendrons sur ce mot-clé dans la partie suivante.

Implémenter une interface explicitement

J'en profite ici pour faire un aparté sur l'implémentation d'interface explicitement.

J'ai choisi délibérément de ne pas le mettre dans le chapitre des interfaces car c'est un cas relativement rare mais qui se produit justement quand on implémente l'interface `IEnumerable<T>`.

Cela vient du fait que l'interface `IEnumerable`, non générique, expose une propriété `Current`. De même, l'interface `IEnumerable<T>`, générique, qui hérite de `IEnumerable`, expose également une propriété `Current`.

Il y a donc une ambiguïté car les deux propriétés portent le même nom, mais ne renvoient pas la même chose. Ce qui est contraire aux règles que nous avons déjà vues. Pour faire la différence, il suffira de préfixer la propriété par le nom de l'interface et de ne pas mettre le mot-clé `public`.

L'implémentation explicite a également un intérêt dans le code suivant :

```
public interface ICarnivore
{
    void Manger();
}

public interface IFrugivore
{
    void Manger();
}

public class Homme : ICarnivore, IFrugivore
{
    public void Manger()
    {
        Console.WriteLine("Je mange");
    }
```

```
}

class Program
{
    static void Main(string[] args)
    {
        Homme homme = new Homme();
        homme.Manger();
        ((ICarnivore)homme).Manger();
        ((IFrugivore)homme).Manger();
    }
}
```

Ici, ce code compile car la classe `Homme` implémente la méthode `Manger` qui est commune aux deux interfaces. Par contre, il n'est pas possible de faire la distinction entre le fait de manger en tant qu'homme, en tant que `ICarnivore` ou en tant que `IFrugivore`.

Ce code affichera :

```
Je mange
Je mange
Je mange
```

Si c'est le comportement attendu, tant mieux. Si ce n'est pas le cas, il va falloir implémenter au moins une des interfaces de manière explicite :

```
public class Homme : ICarnivore, IFrugivore
{
    public void Manger()
    {
        Console.WriteLine("Je mange");
    }

    void IFrugivore.Manger()
    {
        Console.WriteLine("Je mange en tant que IFrugivore");
    }

    void ICarnivore.Manger()
    {
        Console.WriteLine("Je mange en tant que ICarnivore");
    }
}
```

Avec ce code, notre exemple affichera :

```
Je mange
Je mange en tant que ICarnivore
Je mange en tant que IFrugivore
```

Si vous vous rappelez, nous avions vu au moment du chapitre sur les interfaces que Visual C# Express nous proposait de nous aider dans l'implémentation de l'interface. Par le bouton droit, vous aviez également accès au sous-menu `implémenter l'interface explicitement`. Vous pouvez vous en servir dans ce cas précis.

Je m'arrête là sur l'implémentation d'une interface explicite, même s'il y aurait d'autres points à voir. Globalement, en situation réelle, cela ne vous servira jamais.

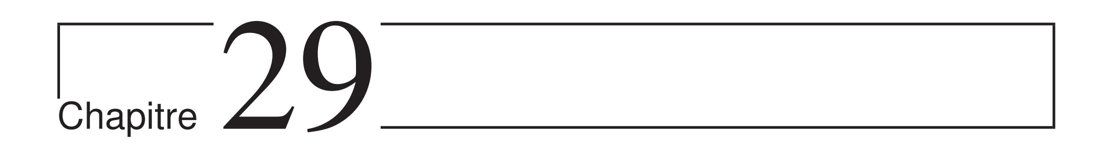
Voilà pour ce TP. Nous avons créé une classe générique permettant de gérer les listes chaînées. Ceci nous a permis de manipuler ces types ô combien indispensables et de nous entraîner à la généricité. Nous en avons même profité pour voir comment faire en sorte qu'une classe soit énumérable, en implémentant la version générique de `IEnumerable`.

Notez bien sûr que cette classe est fonctionnellement incomplète. Il aurait été judicieux de rajouter une méthode permettant de supprimer un élément par exemple. À noter qu'une classe qui fait à peu près le même travail existe dans le framework .NET, elle s'appelle `LinkedList`. Vous trouverez sa documentation via le code web suivant :

▷ Documentation `LinkedList`
Code web : 607852

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 972348



Chapitre 29

Les méthodes d'extension

Difficulté : 

En général, pour ajouter des fonctionnalités à une classe, nous pourrons soit modifier le code source de la classe, soit créer une classe dérivée de notre classe et ajouter ces fonctionnalités.

Dans ce chapitre nous allons voir qu'il existe un autre moyen pour étendre une classe : ceci est possible grâce aux méthodes d'extension. Elles sont intéressantes si nous n'avons pas la main sur le code source de la classe ou si la classe n'est pas dérivable.

Partons à la découverte de ces fameuses méthodes...



Qu'est-ce qu'une méthode d'extension ?

Comme son nom l'indique, une méthode d'extension permet d'étendre une classe en lui rajoutant des méthodes. C'est pratique lorsque nous ne possédons pas le code source d'une classe et qu'il s'avère difficile de la modifier.

D'une manière générale, lorsqu'on souhaite modifier une classe dont on n'a pas le code source, on utilise une classe dérivée ; ce qui est impossible avec des objets qui ne sont pas dérivables, comme par exemple les structures telles que `int` ou `double` ou également avec la classe `String`. En effet, `String` n'est pas dérivable. Nous verrons plus tard comment c'est possible, mais pour l'instant, admettons-le.

Si vous ne me croyez pas, vous pouvez toujours tenter de compiler le code suivant :

```
public class StringEvoluee : String
{
}
```

Créer une méthode d'extension

Si par exemple nous souhaitons créer une méthode permettant de crypter une chaîne de caractères dans un format que nous seuls comprenons, il serait judicieux de créer une méthode dans une classe `StringCryptee` qui dérive de `String`. Comme ceci n'est pas possible, la seule chose qu'il nous reste, c'est de créer une méthode statique utilitaire faisant cet encodage :

```
class Program
{
    static void Main(string[] args)
    {
        string chaineNormale = "Bonjour à tous";
        string chaineCryptee = Encodage.Crypte(chaineNormale);
        Console.WriteLine(chaineCryptee);
        chaineNormale = Encodage.Decrypte(chaineCryptee);
        Console.WriteLine(chaineNormale);
    }

}

public static class Encodage
{
    public static string Crypte(string chaine)
    {
        return Convert.ToBase64String(Encoding.Default.GetBytes
            (chaine));
    }

    public static string Decrypte(string chaine)
    {
```

```
            return Encoding.Default.GetString(Convert.
                FromBase64String(chaine));
        }
}
```

Bon, je vous l'accorde, notre encodage secret ne l'est pas tant que ça ! Il s'avère que j'utilise ici un encodage en base 64, algorithme archiconnu. Mais bon, c'est pour l'exemple !

Utiliser une méthode d'extension

Ces méthodes statiques jouent bien leur rôle. Mais il est possible de faire en sorte que ces deux méthodes deviennent des méthodes d'extension de la classe `String`. Il suffit d'utiliser le mot-clé `this` devant le premier paramètre de la méthode afin de créer une méthode d'extension :

```
public static class Encodage
{
    public static string Crypte(this string chaine)
    {
        return Convert.ToBase64String(Encoding.Default.GetBytes
            (chaine));
    }

    public static string Decrypte(this string chaine)
    {
        return Encoding.Default.GetString(Convert.
            FromBase64String(chaine));
    }
}
```

Nous pourrons désormais utiliser ces méthodes comme si elles faisaient partie de la classe `String` :

```
string chaineNormale = "Bonjour à tous";
string chaineCryptee = chaineNormale.Crypte();
Console.WriteLine(chaineCryptee);
chaineNormale = chaineCryptee.Decrypte();
Console.WriteLine(chaineNormale);
```

Pas mal non ? De plus, si nous regardons dans la complétion automatique, nous pourrons voir apparaître nos fameuses méthodes (voir figure 29.1).

Plutôt pratique. Évidemment, en créant une méthode d'extension, nous n'avons pas accès aux méthodes privées ou variables membres internes à la classe. La preuve, les méthodes d'extension sont des méthodes statiques qui travaillent hors de toute instance de classe.



Ces méthodes doivent donc être statiques et situées à l'intérieur d'une classe statique.

```
static void Main(string[] args)
{
    string chaineNormale = "Bonjour à tous";
    string chaineCryptee = chaineNormale.Crypte();
    Console.WriteLine(chaineCryptee);
    chaineNormale = chaineCryptee.;
    Console.WriteLine(chaineNormal
}
```


FIGURE 29.1 – Notre méthode d'extension apparaît dans la complétion automatique

Par contre, il faut faire attention à l'espace de nom où se situent nos méthodes d'extension. Si le `using` correspondant n'est pas inclus, nous ne verrons pas les méthodes d'extension.

Remarquez que les méthodes d'extension fonctionnent aussi avec les interfaces. Plus précisément, elles viennent étendre toutes les classes qui implémentent une interface. Par exemple, avec deux classes implémentant l'interface `IVolant` :

```
public interface IVolant
{
    void Voler();
}

public class Oiseau : IVolant
{
    public void Voler()
    {
        Console.WriteLine("Je vole");
    }
}

public class Avion : IVolant
{
    public void Voler()
    {
        Console.WriteLine("Je vole");
    }
}
```

Si je crée une méthode d'extension prenant en paramètres un `IVolant`, préfixé par `this` :

```
public static class Extentions
{
    public static void Atterrir(this IVolant volant)
    {
        Console.WriteLine("J'atterris");
```

```
    }
}
```

Je peux ensuite accéder à cette méthode pour les objets `Avion` et `Oiseau` :

```
Avion a = new Avion();
Oiseau b = new Oiseau();
a.Atterrir();
b.Atterrir();
```

À noter que le framework .NET se sert de ceci pour proposer un grand nombre de méthodes d’extension sur les objets implémentant `IEnumerable` ; nous les étudierons un peu plus tard.

En résumé

– Une méthode d’extension permet d’étendre une classe en lui rajoutant des méthodes.
– Il n’est pas recommandé d’utiliser des méthodes d’extension lorsqu’on dispose déjà du code source de la classe ou qu’on peut facilement en créer un type dérivé.
– On utilise le mot-clé `this` en premier paramètre d’une classe statique pour étendre une classe.

Chapitre 30

Délégués, événements et expressions lambdas

Difficulté : 

Dans ce chapitre, nous allons aborder les délégués, les événements et les expressions lambdas. Les délégués et les événements sont des types du framework .NET que nous n'avons pas encore vus. Ils permettent d'adresser des solutions notamment dans le cadre d'une programmation par événements, comme c'est le cas lorsque nous réalisons des applications nécessitant de réagir à une action faite par un utilisateur. Nous verrons dans ce chapitre que les expressions lambdas vont de pair avec les délégués.



Les délégués (delegate)

Les délégués [1] en C# ne s'occupent pas de la classe, ni du personnel ! Ils permettent de créer des variables spéciales. Ce sont des variables qui « pointent » vers une méthode. C'est un peu comme les pointeurs de fonctions en C ou C++, sauf qu'ici on sait exactement ce que l'on utilise car le C# est fortement typé.

Le délégué va nous permettre de définir une signature de méthode et avec lui, nous pourrons pointer vers n'importe quelle méthode qui respecte cette signature.

En général, on utilise un délégué quand on veut passer une méthode en paramètre d'une autre méthode. Un petit exemple sera sans doute plus parlant qu'un long discours. Ainsi, le code suivant :

```
public class TrieurDeTableau
{
    private delegate void DelegateTri(int[] tableau);
}
```

crée un délégué privé à la classe `TrieurDeTableau` qui permettra de pointer vers des méthodes qui ne retournent rien (`void`) et qui acceptent un tableau d'entiers en paramètre.

C'est justement le cas des méthodes `TriAscendant()` et `TriDescendant()` que nous allons ajouter à la classe (ça tombe bien !) :

```
public class TrieurDeTableau
{
    private delegate void DelegateTri(int[] tableau);

    private void TriAscendant(int[] tableau)
    {
        Array.Sort(tableau);
    }

    private void TriDescendant(int[] tableau)
    {
        Array.Sort(tableau);
        Array.Reverse(tableau);
    }
}
```

Vous aurez compris que la méthode `TriAscendant` utilise la méthode `Array.Sort` pour trier un tableau par ordre croissant. Inversement, la méthode `TriDescendant()` trie le tableau par ordre décroissant en triant par ordre croissant et en inversant ensuite le tableau.

Il ne reste plus qu'à créer une méthode dans la classe permettant d'utiliser le tri ascendant et le tri descendant, grâce à notre délégué :

```
public class TrieurDeTableau
```

1. En anglais, *delegate*.

```
{
    [...Code supprimé pour plus de clarté...]

    public void DemoTri(int[] tableau)
    {
        DelegateTri tri = TriAscendant;
        tri(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }

        Console.WriteLine();
        tri = TriDescendant;
        tri(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
    }
}
```

Nous voyons ici que dans la méthode `DemoTri` nous commençons par déclarer une variable du type du délégué `DelegateTri`, qui est le délégué que nous avons créé. Puis nous faisons pointer cette variable vers la méthode `TriAscendant()`.



Nul besoin ici d'utiliser les parenthèses, mais juste le nom de la méthode. Il s'agit juste d'une affectation.

Nous invoquons ensuite la méthode `TriAscendant()` à travers la variable qui va permettre de trier le tableau par ordre croissant avant d'afficher son contenu. Cette fois-ci, il faut bien sûr utiliser les parenthèses car nous invoquons la méthode. Puis nous faisons pointer la variable vers la méthode `TriDescendant()` qui va nous permettre de faire la même chose mais avec un tri décroissant.

Nous pouvons appeler cette classe de cette façon :

```
static void Main(string[] args)
{
    int[] tableau = new int[] { 4, 1, 6, 10, 8, 5 };
    new TrieurDeTableau().DemoTri(tableau);
}
```

Notre code affichera au final les entiers triés par ordre croissant, puis les mêmes entiers triés par ordre décroissant.



Ok, mais pourquoi utiliser ce délégué ? On pourrait très bien appeler d'abord la méthode de tri ascendant et ensuite la méthode de tri descendant. Comme on l'a toujours fait !

Eh bien, l'intérêt ici est que le délégué est très souple et va permettre de réorganiser le code (on parle également de **refactoriser du code**). Ainsi, en rajoutant la méthode suivante dans la classe :

```
private void TrierEtAfficher(int[] tableau, DelegateTri
    methodeDeTri)
{
    methodeDeTri(tableau);
    foreach (int i in tableau)
    {
        Console.WriteLine(i);
    }
}
```

nous pourrons grandement simplifier la méthode `DemoTri` :

```
public void DemoTri(int[] tableau)
{
    TrierEtAfficher(tableau, TriAscendant);
    Console.WriteLine();
    TrierEtAfficher(tableau, TriDescendant);
}
```

Ce qui produira le même résultat que précédemment. Qu'avons-nous fait ici ?

Nous avons utilisé le délégué comme paramètre d'une méthode. Ce délégué est ensuite utilisé pour invoquer une méthode que nous aurons passée en paramètres. C'est ce que nous faisons en disant d'utiliser la méthode `TrierEtAfficher` une première fois avec la méthode `TriAscendant()` et une deuxième fois avec la méthode `TriDescendant()`.

Plutôt pas mal non ? Il est même possible de définir la méthode qui sera utilisée à l'intérieur de `TrierEtAfficher()` sans avoir à l'écrire complètement dans le corps de la classe.

Cela peut être utile si la méthode n'est vouée à être utilisée que dans cette unique situation et qu'elle n'est jamais appelée à un autre endroit. Par exemple, plutôt que de définir complètement la méthode `TriAscendant()`, je pourrais la définir directement au moment de l'appel de la méthode :

```
public class TrieurDeTableau
{
    private delegate void DelegateTri(int[] tableau);

    private void TrierEtAfficher(int[] tableau, DelegateTri
        methodeDeTri)
    {
        methodeDeTri(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
    }
```

```

    public void DemoTri(int[] tableau)
    {
        TrierEtAfficher(tableau, delegate(int[] leTableau)
        {
            Array.Sort(leTableau);
        });

        Console.WriteLine();

        TrierEtAfficher(tableau, delegate(int[] leTableau)
        {
            Array.Sort(tableau);
            Array.Reverse(tableau);
        });

    }
}
```

Ainsi, je n'aurai plus besoin des méthodes `TriAscendant()` et `TriDescendant()`.

Le fait de définir la méthode directement au niveau du paramètre d'appel est ce qu'on appelle « **utiliser une méthode anonyme** ». Anonyme car la méthode n'a pas de nom. Elle n'a de vie qu'à cet endroit-là.

La syntaxe est un peu particulière, mais au lieu d'utiliser une variable de type `delegate` qui pointe vers une méthode, c'est comme si on écrivait directement la méthode.

On utilise le mot-clé `delegate` suivi de la déclaration du paramètre. Évidemment, le délégué anonyme doit respecter la signature de `DelegateTri` que nous avons défini plus haut. Enfin, nous faisons suivre avec un bloc de code qui correspond au corps de la méthode anonyme.



À noter que le fait d'utiliser le mot-clé `delegate` revient en fait à créer une classe qui dérive de `System.Delegate` et qui implémente la logique de base d'un délégué. Le C# nous masque tout ceci afin d'être le plus efficace possible.

Diffusion multiple, le multicast

Il faut également savoir que le délégué peut être multicast, cela veut dire qu'il peut pointer vers plusieurs méthodes. Améliorons le premier exemple :

```
public class TrieurDeTableau
{
    private delegate void DelegateTri(int[] tableau);

    private void TriAscendant(int[] tableau)
```

```
    {
        Array.Sort(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine();
    }

    private void TriDescendant(int[] tableau)
    {
        Array.Sort(tableau);
        Array.Reverse(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
    }

    public void DemoTri(int[] tableau)
    {
        DelegateTri tri = TriAscendant;
        tri += TriDescendant;
        tri(tableau);
    }
}
```

Ici, j'utilise `Console.WriteLine` directement dans chaque méthode de tri afin de bien voir le résultat du tri du tableau. L'important est de voir que dans la méthode `DemoTri`, je commence par créer un délégué que je fais pointer vers la méthode `TriAscendant`. Puis j'ajoute à ce délégué, avec l'opérateur +=, une nouvelle méthode, à savoir `TriDescendant`. Désormais, le fait d'invoquer le délégué va invoquer les deux méthodes ; ce qui produira en sortie :

```
1
4
5
6
8
10

10
8
6
5
4
1
```

Ce détail prend toute son importance avec les événements que nous verrons plus loin.

À noter que le résultat de ce code est évidemment identique en utilisant les méthodes anonymes :

```
public void DemoTri(int[] tableau)
{
    DelegateTri tri = delegate(int[] leTableau)
    {
        Array.Sort(leTableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine();
    };
    tri += delegate(int[] leTableau)
    {
        Array.Sort(tableau);
        Array.Reverse(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
    };
    tri(tableau);
}
```



Il faut quand même remarquer que l'ordre dans lequel sont appelées les méthodes n'est pas garanti et ne dépend pas forcément de l'ordre dans lequel nous les avons ajoutées au délégué.

Les délégués génériques `Action` et `Func`



C'est très bien tout ça, mais cela veut dire qu'à chaque fois que je vais avoir besoin d'utiliser un délégué, je vais devoir créer un nouveau type en utilisant le mot-clé `delegate` ?

Pas forcément, c'est là qu'interviennent les délégués génériques `Action` et `Func`. `Action` est un délégué qui permet de pointer vers une méthode qui ne renvoie rien et qui peut accepter jusqu'à 16 types différents. Cela veut dire que le code précédent peut être remplacé par :

```
public class TrieurDeTableau
{
    private void TrierEtAfficher(int[] tableau, Action<int[]>
        methodeDeTri)
    {
```

```
        methodeDeTri(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
    }

    public void DemoTri(int[] tableau)
    {
        TrierEtAfficher(tableau, delegate(int[] leTableau)
        {
            Array.Sort(leTableau);
        });

        Console.WriteLine();

        TrierEtAfficher(tableau, delegate(int[] leTableau)
        {
            Array.Sort(tableau);
            Array.Reverse(tableau);
        });
    }
}
```

Notez que la différence se situe au niveau du paramètre de la méthode `TrierEtAfficher` qui prend un `Action<int[]>`. En fait, cela est équivalent à créer un délégué qui ne renvoie rien et qui prend un tableau d'entiers en paramètre. Si notre méthode avait deux paramètres, il aurait suffi d'utiliser la forme de `Action` avec plusieurs paramètres génériques, par exemple `Action<int[], string>` pour avoir une méthode qui ne renvoie rien et qui prend un tableau d'entiers et une chaîne de caractères en paramètres.

Lorsque la méthode renvoie quelque chose, on peut utiliser le délégué `Func<T>`, sachant qu'ici, c'est le dernier paramètre générique qui sera du type de retour du délégué. Par exemple :

```
public class Operations
{
    public void DemoOperations()
    {
        double division = Calcul(delegate(int a, int b)
        {
            return (double)a / (double)b;
        }, 4, 5);

        double puissance = Calcul(delegate(int a, int b)
        {
            return Math.Pow((double)a, (double)b);
        }, 4, 5);

        Console.WriteLine("Division : " + division);
        Console.WriteLine("Puissance : " + puissance);
```

```
    }

    private double Calcul(Func<int, int, double>
        methodeDeCalcul, int a, int b)
    {
        return methodeDeCalcul(a, b);
    }
}
```

Ici, dans la méthode `Calcul`, on utilise le délégué `Func` pour indiquer que la méthode prend deux entiers en paramètres et renvoie un `double`. Si nous utilisons cette classe avec le code suivant :

```
class Program
{
    static void Main(string[] args)
    {
        new Operations().DemoOperations();
    }
}
```

nous aurons :

```
Division : 0,8
Puissance : 1024
```

Les expressions lambdas

Non, il ne s'agit pas d'une expression qui danse la *lambada*, mais d'une façon simplifiée d'écrire les délégués que nous avons vus au-dessus !

Ainsi, le code suivant :

```
DelegateTri tri = delegate(int[] leTableau)
{
    Array.Sort(leTableau);
};
```

peut également s'écrire de cette façon :

```
DelegateTri tri = (leTableau) =>
{
    Array.Sort(leTableau);
};
```

Cette syntaxe est particulière. La variable `leTableau` permet de spécifier le paramètre d'entrée de l'expression lambda. Ce paramètre est écrit entre parenthèses. Ici, pas besoin d'indiquer son type vu qu'il est connu par la signature associée au délégué. On utilise ensuite la flèche => pour définir l'expression lambda qui sera utilisée. Elle s'écrit de la même façon qu'une méthode, dans un bloc de code.

L'expression lambda `(leTableau) =>` se lit : « `leTableau` conduit à ».

Dans le corps de la méthode, nous voyons que nous utilisons la variable `leTableau` de la même façon que précédemment.

Dans ce cas précis, il est encore possible de raccourcir l'écriture car la méthode ne contient qu'une seule ligne ; on pourra alors l'écrire de cette façon :

```
TrierEtAfficher(tableau, leTableau => Array.Sort(leTableau));
```



S'il y a un seul paramètre à l'expression lambda, on peut omettre les parenthèses.

Quand il y a deux paramètres, on les sépare par une virgule. À noter qu'on n'indique nulle part le type de retour, s'il y en a un. Notre expression lambda remplaçant le calcul de la division peut donc s'écrire ainsi :

```
double division = Calcul((a, b) =>
{
    return (double)a / (double)b;
}, 4, 5);
```

Lorsque l'instruction possède une unique ligne, on peut encore en simplifier l'écriture ; ce qui donne :

```
double division = Calcul((a, b) => (double)a / (double)b, 4, 5)
    ;
```



Pourquoi tout ce blabla sur les delegate et les expressions lambdas ?

Pour deux raisons :

– Parce que les délégués sont la base des événements.
– À cause des méthodes d'extension LINQ.

Nous parlerons dans la partie suivante des méthodes d'extension LINQ. Quant aux événements, explorons-les dès maintenant !

Les événements

Les événements sont un mécanisme du C# permettant à une classe d'être notifiée d'un changement. Par exemple, on peut vouloir s'abonner à un changement de prix d'une voiture. La base des événements est le délégué. On pourra stocker dans un événement un ou plusieurs délégués qui pointent vers des méthodes respectant la signature de l'événement.

Un événement est défini grâce au mot-clé `event`. Prenons cet exemple :

```
public class Voiture
{
    public delegate void DelegateDeChangementDePrix(decimal nouveauPrix);
    public event DelegateDeChangementDePrix ChangementDePrix;
    public decimal Prix { get; set; }

    public void PromoSurLePrix()
    {
        Prix = Prix / 2;
        if (ChangementDePrix != null)
            ChangementDePrix(Prix);
    }
}
```

Dans la classe `Voiture`, nous définissons un délégué qui ne retourne rien et qui prend en paramètre un décimal. Nous définissons ensuite un événement basé sur ce délégué, avec, comme nous l'avons vu, l'utilisation du mot-clé `event`. Enfin, dans la méthode de promotion, après un changement de prix (division par 2), nous notifions les éventuels objets qui se seraient abonnés à cet événement en invoquant l'événement et en lui fournissant en paramètre le nouveau prix.

À noter que nous testons d'abord s'il y a un abonné à l'événement (en testant s'il est différent de `null`) avant de le lever.

Pour s'abonner à cet événement, il suffit d'utiliser le code suivant :

```
class Program
{
    static void Main(string[] args)
    {
        new DemoEvenement().Demo();
    }
}

public class DemoEvenement
{
    public void Demo()
    {
        Voiture voiture = new Voiture { Prix = 10000 };

        Voiture.DelegateDeChangementDePrix delegateChangementDePrix = voiture_ChangementDePrix;
        voiture.ChangementDePrix += delegateChangementDePrix;

        voiture.PromoSurLePrix();
    }

    private void voiture_ChangementDePrix(decimal nouveauPrix)
    {
        Console.WriteLine("Le nouveau prix est de : " +
```

```
                nouveauPrix);
        }
}
```

Nous créons une voiture, et nous créons un délégué du même type que l'événement. Nous le faisons pointer vers une méthode qui respecte la signature du délégué. Ainsi, à chaque changement de prix, la méthode `voiture_ChangementDePrix` va être appelée et le paramètre `nouveauPrix` possédera le nouveau prix qui vient d'être calculé.

Appelons la promotion en invoquant la méthode `ChangementDePrix()`. Nous pouvons nous rendre compte que l'application nous affiche le nouveau prix qui est l'ancien divisé par 2.

Lorsque nous commençons à écrire le code qui va permettre de nous abonner à l'événement, la complétion automatique nous propose facilement de créer une méthode qui respecte la signature de l'événement. Il suffit de saisir l'événement, d'ajouter ensuite un += et d'appuyer sur la touche Tab pour que Visual C# Express nous propose de tout insérer automatiquement (voir la figure 30.1).

FIGURE 30.1 – Complétion automatique d'événement

Ce qui génère le code suivant :

```
voiture.ChangementDePrix += new Voiture.
    DelegateDeChangementDePrix(voiture_ChangementDePrix);
```

ainsi que la méthode :

```
void voiture_ChangementDePrix(decimal nouveauPrix)
{
    throw new NotImplementedException();
}
```

On peut aisément simplifier l'abonnement avec :

```
voiture.ChangementDePrix += voiture_ChangementDePrix;
```

comme on l'a déjà vu. Notez que vous pouvez également rajouter la visibilité `private` sur la méthode générée afin que cela soit plus explicite.

```
private void voiture_ChangementDePrix(decimal nouveauPrix)
```

```
{
}
```

L'utilisation du += permet d'ajouter un nouveau délégué à l'événement. Il sera éventuellement possible d'ajouter un autre délégué avec le même opérateur ; ainsi deux méthodes seront désormais notifiées en cas de changement de prix.

Inversement, il est possible de se désabonner d'un événement en utilisant l'opérateur -=.

Les événements sont largement utilisés dans les applications ayant recours au C#, comme les applications clients lourds développées avec WPF par exemple. Ce sont des applications comme un traitement de texte ou un navigateur internet. Par exemple, lorsque l'on clique sur un bouton, un événement est levé.

Ces événements utilisent en général une construction à base du délégué `EventHandler` ou sa version générique `EventHandler<>`. Ce délégué accepte deux paramètres. Le premier de type `object` qui représente la source de l'événement, c'est-à-dire l'objet qui a levé l'événement. Le second est une classe qui dérive de la classe de base `EventArgs`.

Réécrivons notre exemple avec ce nouveau handler. Nous avons donc besoin en premier lieu d'une classe qui dérive de la classe `EventArgs` :

```
public class ChangementDePrixEventArgs : EventArgs
{
    public decimal Prix { get; set; }
}
```

Plus besoin de déclaration de délégué, nous utilisons directement `EventHandler` dans notre classe `Voiture` :

```
public class Voiture
{
    public event EventHandler<ChangementDePrixEventArgs>
        ChangementDePrix;
    public decimal Prix { get; set; }

    public void PromoSurLePrix()
    {
        Prix = Prix / 2;
        if (ChangementDePrix != null)
            ChangementDePrix(this, new
                ChangementDePrixEventArgs { Prix = Prix });
    }
}
```

Et notre démo devient :

```
class Program
{
    static void Main(string[] args)
    {
        new DemoEvenement().Demo();
```

```
    }
}

public class DemoEvenement
{
    public void Demo()
    {
        Voiture voiture = new Voiture { Prix = 10000 };

        voiture.ChangementDePrix += voiture_ChangementDePrix;

        voiture.PromoSurLePrix();
    }

    private void voiture_ChangementDePrix(object sender,
        ChangementDePrixEventArgs e)
    {
        Console.WriteLine("Le nouveau prix est de : " + e.Prix)
            ;
    }
}
```

Remarquez que la méthode `voiture_ChangementDePrix` prend désormais deux paramètres. Le premier représente l'objet `Voiture` ; si nous en avions besoin, nous pourrions l'utiliser avec un cast adéquat. Le second représente l'objet contenant le prix de la voiture en promotion.

À noter que, de manière générale, nous allons surtout utiliser les événements définis par le framework .NET. Il sera donc assez rare d'avoir à en définir un soi-même.

En résumé

- Les délégués permettent de créer des variables pointant vers des méthodes.
- Les délégués sont à la base des événements.
- On utilise les expressions lambdas pour simplifier l'écriture des délégués.
- Les événements sont un mécanisme du C# permettant à une classe d'être notifiée d'un changement.

Chapitre 31

Gérer les erreurs : les exceptions

Difficulté : 

Nous avons parlé rapidement des erreurs dans nos applications C# en disant qu'il s'agissait d'exceptions. C'est le moment d'en savoir un peu plus et surtout d'apprendre à les gérer ! Dans ce chapitre, nous allons apprendre comment créer et intercepter une exception.



Intercepter une exception

Rappelez-vous de ce code :

```
string chaine = "dix";
int valeur = Convert.ToInt32(chaine);
```

Si nous l'exécutons, nous aurons un message d'erreur :

```
Exception non gérée : System.FormatException: Le format de la
    chaîne d'entrée est incorrect.
à System.Number.StringToNumber(String str, NumberStyles options,
     NumberBuffer
& number, NumberFormatInfo info, Boolean parseDecimal)
à System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
à System.Convert.ToInt32(String value)
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    14
```

L'application nous affiche un message d'erreur et plante lamentablement, produisant par la même occasion un rapport d'erreur. Ce qui se passe en fait, c'est que lors de la conversion, si le framework .NET n'arrive pas à convertir correctement la chaîne de caractères en entier, **il lève une exception**. Cela veut dire qu'il informe le programme qu'il rencontre un cas limite qui nécessite d'être géré. Si ce cas limite n'est pas géré, alors l'application plante et c'est le CLR qui intercepte l'erreur et fait produire un rapport au système d'exploitation.



Pourquoi une exception et pas un message d'erreur ?

L'intérêt des exceptions est qu'elles sont typées. C'en est fini des codes d'erreurs incompréhensibles. C'est le type de l'exception, c'est-à-dire sa classe, qui va nous permettre d'identifier le problème.

Pour éviter le plantage de l'application, nous devons gérer ces cas limites et intercepter les exceptions. Pour ce faire, il faut encadrer les instructions pouvant atteindre des cas limites avec le bloc d'instruction `try...catch`, par exemple :

```
try
{
    string chaine = "dix";
    int valeur = Convert.ToInt32(chaine);
    Console.WriteLine("Ce code ne sera jamais affiché");
}
catch (Exception)
{
```

```
    Console.WriteLine("Une erreur s'est produite dans la
        tentative de conversion");
}
```

Si nous exécutons ce bout de code, l'application ne plantera plus et affichera qu'une erreur s'est produite. . .

Nous avons « attrapé » l'exception levée par la méthode de conversion grâce au mot-clé `catch`. Cette construction nous permet de surveiller l'exécution d'un bout de code, situé dans le bloc `try` et, s'il y a une erreur, alors nous interrompons son exécution pour traiter l'erreur en allant dans le bloc `catch`. La suite du code dans le `try`, à savoir l'affichage de la ligne avec `Console.WriteLine`, ne sera jamais exécuté, car lorsque la conversion échoue, il saute directement au bloc `catch`. Inversement, il est possible de ne jamais passer dans le bloc `catch` si les instructions ne provoquent pas d'erreur :

```
try
{
    string chaine = "10";
    int valeur = Convert.ToInt32(chaine);
    Console.WriteLine("Conversion OK");
}
catch (Exception)
{
    Console.WriteLine("Nous ne passons jamais ici ...");
}
```

Ainsi le code ci-dessus affichera uniquement que la conversion est bonne et, en toute logique, il ne passera pas dans le bloc de traitement d'erreur, car il n'y en a pas eu. Il est possible d'obtenir des informations sur l'exception en utilisant un paramètre dans le bloc `catch`. Ce paramètre est une variable du type `Exception`, qui dans l'exemple suivant s'appelle `ex` :

```
try
{
    string chaine = "dix";
    int valeur = Convert.ToInt32(chaine);
}
catch (Exception ex)
{
    Console.WriteLine("Il y a un eu une erreur, plus d'
        informations ci-dessous :");
    Console.WriteLine();
    Console.WriteLine("Message d'erreur : " + ex.Message);
    Console.WriteLine();
    Console.WriteLine("Pile d'appel : " + ex.StackTrace);
    Console.WriteLine();
    Console.WriteLine("Type de l'exception : " + ex.GetType());
}
```

Ici, nous affichons le message d'erreur, la pile d'appel et le type de l'exception, ce qui donne le message suivant :

```
Il y a un eu une erreur, plus d'informations ci-dessous :

Message d'erreur : Le format de la chaîne d'entrée est incorrect
    .

Pile d'appel : à System.Number.StringToNumber(String str,
    NumberStyles options, NumberBuffer& number, NumberFormatInfo
    info, Boolean parseDecimal)
à System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
à System.Convert.ToInt32(String value)
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    16

Type de l'exception : System.FormatException
```

D'une manière générale, la méthode `ToString()` de l'exception fournit des informations suffisantes pour identifier l'erreur :

```
try
{
    string chaine = "dix";
    int valeur = Convert.ToInt32(chaine);
}
catch (Exception ex)
{
    Console.WriteLine("Il y a un eu une erreur : " + ex.
        ToString());
}
```

Ce code affichera :

```
Il y a un eu une erreur : System.FormatException: Le format de
    la chaîne d'entrée est incorrect.
à System.Number.StringToNumber(String str, NumberStyles options,
     NumberBuffer
& number, NumberFormatInfo info, Boolean parseDecimal)
à System.Number.ParseInt32(String s, NumberStyles style,
    NumberFormatInfo info)
à System.Convert.ToInt32(String value)
à MaPremiereApplication.Program.Main(String[] args) dans C:\
    Users\Nico\Documents\Visual Studio 2010\Projects\C#\
    MaPremiereApplication\MaPremiereApplication\Program.cs:ligne
    16
```

Les exceptions peuvent être de beaucoup de formes. Ici nous remarquons que l'exception est de type `System.FormatException`. Cette exception est utilisée en général lorsque

le format d'un paramètre ne correspond pas à ce qui est attendu. En l'occurrence ici, nous attendons un paramètre de type chaîne de caractères qui représente un entier. C'est une exception spécifique qui est dédiée à un type d'erreur précis.

Il faut savoir que comme beaucoup d'autres objets du framework .NET, les exceptions spécifiques dérivent d'une classe de base, à savoir la classe `Exception`, dont vous pouvez trouver la documentation via le code web suivant :

▷ Exceptions
Code web : 368326

Il existe une hiérarchie entre les exceptions. Globalement, deux grandes familles d'exceptions existent : `ApplicationException` et `SystemException`. Elles dérivent toutes les deux de la classe de base `Exception`. La première est utilisée lorsque des erreurs récupérables sur des applications apparaissent ; la seconde est utilisée pour toutes les exceptions générées par le framework .NET. Par exemple, l'exception que nous avons vue, `FormatException` dérive directement de `SystemException`, qui dérive elle-même de la classe `Exception`.

Le framework .NET dispose de beaucoup d'exceptions correspondant à beaucoup de situations. Notons encore au passage une autre exception bien connue des développeurs qui est la `NullReferenceException`. Elle se produit lorsqu'on essaie d'accéder à un objet qui vaut `null`. Par exemple :

```
Voiture voiture = null;
voiture.Vitesse = 10;
```

Vous aurez remarqué que dans la construction suivante :

```
try
{
    string chaine = "dix";
    int valeur = Convert.ToInt32(chaine);
}
catch (Exception ex)
{
    Console.WriteLine("Il y a un eu une erreur : " + ex.
        ToString());
}
```

nous voyons que le bloc `catch` prend en paramètre la classe de base `Exception`. Cela veut dire que nous souhaitons intercepter toutes les exceptions qui dérivent de `Exception` ; c'est-à-dire en fait toutes les exceptions, car toute exception dérive forcément de la classe `Exception`. C'est utile lorsque nous voulons attraper toutes les exceptions. Mais savons-nous forcément quoi faire dans le cas de toutes les erreurs ? Il est possible d'être plus précis afin de n'attraper qu'un seul type d'exception. Il suffit de préciser le type de l'exception attendue comme paramètre du `catch`. Par exemple le code suivant nous permet d'intercepter toutes les exceptions du type `FormatException` :

```
try
{
    string chaine = "dix";
```

```
    int valeur = Convert.ToInt32(chaine);
}
catch (FormatException ex)
{
    Console.WriteLine(ex);
}
```

Cela veut par contre dire que si nous avons une autre exception à ce moment-là, du style `NullReferenceException`, elle ne sera pas attrapée ; ce qui fait que le code suivant va planter :

```
try
{
    Voiture v = null;
    v.Vitesse = 10;
}
catch (FormatException ex)
{
    Console.WriteLine("Erreur de format : " + ex);
}
```

En effet, nous demandons uniquement la surveillance de l'exception `FormatException`. Ainsi, l'exception `NullReferenceException` ne sera pas attrapée.

Intercepter plusieurs exceptions

Pour attraper les deux exceptions, il est possible d'enchaîner les blocs `catch` avec des paramètres différents :

```
try
{
    // code provoquant une exception
}
catch (FormatException ex)
{
    Console.WriteLine("Erreur de format : " + ex);
}
catch (NullReferenceException ex)
{
    Console.WriteLine("Erreur de référence nulle : " + ex);
}
```

Dans ce code, cela veut dire que si le code surveillé provoque une `FormatException`, alors nous afficherons le message « Erreur de format... » ; s'il provoque une `NullReferenceException`, alors nous afficherons le message « Erreur de référence nulle... ». Notez bien que nous ne pouvons rentrer à chaque fois que dans un seul bloc `catch`. Une autre solution serait d'attraper une exception plus généraliste par exemple `SystemException` dont dérive `FormatException` et `NullReferenceException` :

```
try
{
    // code provoquant une exception
}
catch (SystemException ex)
{
    Console.WriteLine("Erreur système : " + ex);
}
```

Par contre, le code précédent attrape toutes les exceptions dérivant de `SystemException`. C'est le cas de `FormatException` et `NullReferenceException`, mais c'est aussi le cas pour beaucoup d'autres exceptions. Lorsqu'on surveille un bloc de code, on commence en général par surveiller les exceptions les plus fines possible et on remonte en considérant les exceptions les plus générales, jusqu'à terminer par la classe `Exception` :

```
try
{
    // code provoquant une exception
}
catch (FormatException ex)
{
    Console.WriteLine("Erreur de format : " + ex);
}
catch (NullReferenceException ex)
{
    Console.WriteLine("Erreur de référence nulle : " + ex);
}
catch (SystemException ex)
{
    Console.WriteLine("Erreur système autres que
        FormatException et NullReferenceException : " + ex);
}
catch(Exception ex)
{
    Console.WriteLine("Toutes les autres exceptions : " + ex);
}
```

À chaque exécution, c'est le bloc `catch` qui se rapproche le plus de l'exception levée qui est utilisé. C'est un peu comme une opération conditionnelle. Le programme vérifie dans un premier temps que l'exception n'est pas une `FormatException`. Si ce n'est pas le cas, il vérifiera qu'il n'a pas à faire à une `NullReferenceException`. Ensuite, il vérifiera qu'il ne s'agit pas d'une `SystemException`. Enfin, il interceptera toutes les exceptions dans le dernier bloc de code, car `Exception` étant la classe mère, toutes les exceptions sont interceptées avec ce type.

À noter qu'il est possible d'imbriquer les `try...catch` si cela s'avère pertinent. Par exemple :

```
string saisie = Console.ReadLine();
try
{
```

```
    int entier = Convert.ToInt32(saisie);
    Console.WriteLine("La saisie est un entier");
}
catch (FormatException)
{
    try
    {
        double d = Convert.ToDouble(saisie);
        Console.WriteLine("La saisie est un double");
    }
    catch (FormatException)
    {
        Console.WriteLine("La saisie n'est ni un entier, ni un
            double");
    }
}
```

Ce code nous permet de tester si la saisie est un entier. Si une exception se produit, alors nous tentons de le convertir en double. S'il y a encore une exception, alors nous affichons un message indiquant que les deux conversions ont échoué.

Lever une exception

Il est possible de déclencher soi-même la levée d'une exception. C'est utile si nous considérons que notre code a atteint un cas limite, qu'il soit fonctionnel ou technique. Pour lever une exception, nous utilisons le mot-clé `throw`, suivi de l'instance d'une exception. Imaginons par exemple une méthode permettant de calculer la racine carrée d'un double. Nous pouvons obtenir un cas limite lorsque nous tentons de passer un double négatif :

```
public static double RacineCarree(double valeur)
{
    if (valeur <= 0)
        throw new ArgumentOutOfRangeException("valeur", "Le
            paramètre doit être positif");
    return Math.Sqrt(valeur);
}
```

Ici, nous instancions une `ArgumentOutOfRangeException` en utilisant un constructeur à deux paramètres. Ceux-ci permettent d'indiquer le nom du paramètre ainsi que le message d'erreur. Cette exception est une exception du framework .NET utilisée pour indiquer qu'un paramètre est en dehors des plages de valeurs autorisées. C'est exactement ce qu'il nous faut ici. Puis, nous levons l'exception avec le mot-clé `throw`. Nous pouvons utiliser la méthode ainsi :

```
static void Main(string[] args)
{
    try
```

```
    {
        double racine = RacineCarree(-5);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Impossible d'effectuer le calcul : "
            + ex.Message);
    }
}
```

Ce qui affiche :

```
Impossible d’effectuer le calcul : Le paramètre doit être
   positif
Nom du paramètre : valeur
```

Il aurait été possible de lever une exception plus générique avec la classe de base `Exception` :

```
throw new Exception("Le paramètre doit être positif");
```

Mais n’oubliez pas que plus l’exception est finement typée, plus elle sera facile à traiter précisément. Cela permet d’éviter d’attraper toutes les exceptions dans le même `catch` avec la classe de base `Exception`. À noter que lorsque notre programme rencontre le mot-clé `throw`, il s’arrête pour partir dans le bloc `try...catch` correspondant (s’il existe). Cela signifie qu’une méthode qui doit renvoyer un paramètre pourra compiler si son chemin se termine par une levée d’exception, comme c’est le cas pour le calcul de la racine carrée.

Propagation de l’exception

Il est important de noter que lorsqu’un bout de code se situe dans un bloc `try...catch`, tout le code qui est dessous est surveillé, même s’il y a plusieurs méthodes qui s’appellent les unes les autres. Par exemple, si nous appelons depuis la méthode `Main()` une `Methode1()`, qui appelle une `Methode2()`, qui appelle une `Methode3()`, qui lève une exception, alors nous serons capables de l’intercepter depuis la méthode `Main()` avec un `try...catch` :

```
static void Main(string[] args)
{
    try
    {
        Methode1();
    }
    catch (NotImplementedException)
    {
        Console.WriteLine("On intercepte l'exception de la mé
            thode 3");
```

```
    }
}

public static void Methode1()
{
    Methode2();
}

public static void Methode2()
{
    Methode3();
}

public static void Methode3()
{
    throw new NotImplementedException();
}
```

À noter qu'une `NotImplementedException` est une exception utilisée pour indiquer qu'un bout de code n'a pas encore été implémenté. Il est également possible d'attraper une exception, de la traiter et de choisir qu'elle continue à se propager. Par exemple, imaginons que nous avons un bloc `try...catch` qui nous permet de surveiller tout notre programme et que nous ayons à surveiller un bout de code ailleurs dans le programme qui peut produire une situation limite :

```
static void Main(string[] args)
{
    try
    {
        MaMethode();
    }
    catch (Exception ex)
    {
        // ici, on intercepte toutes les erreurs possibles en indiquant qu'un problème inattendu s'est produit
        Console.WriteLine("L'application a rencontré un problème, un mail a été envoyé à l'administrateur ...");
        EnvoyerExceptionAdministrateur(ex);
    }
}

public static void MaMethode()
{
    try
    {
        Console.WriteLine("Veuillez saisir un entier :");
        string chaine = Console.ReadLine();
        int entier = Convert.ToInt32(chaine);
    }
    catch (FormatException)
```

```
    {
        Console.WriteLine("La saisie n'est pas un entier");
    }
    catch (Exception ex)
    {
        EnregistrerErreurDansUnFichierDeLog(ex);
        throw;
    }
}
```

J'ai une saisie à faire et à convertir en entier. Si la conversion échoue, je suis capable de l'indiquer à l'utilisateur en surveillant la `FormatException`. Par contre, si une exception inattendue se produit, je souhaite pouvoir faire quelque chose, en l'occurrence enregistrer l'exception dans un fichier ; mais comme c'est un cas limite non prévu je souhaite que l'exception continue à se propager afin qu'elle soit attrapée par le bloc `try...catch` qui permettra d'envoyer un mail à l'administrateur. Dans ce cas, j'utilise un `catch` généraliste pour traiter les exceptions inattendues afin de loguer l'exception, puis j'utilise directement le mot-clé `throw` afin de permettre de relever l'exception.

Créer une exception personnalisée

Grâce au typage fort des exceptions, il est pratique d'utiliser un type d'exception pour reconnaître un cas limite, comme une erreur de conversion ou une exception de référence nulle. Nous allons pouvoir utiliser certaines de ces exceptions pour nos besoins, comme ce que nous avions fait avec l'exception `ArgumentOutOfRangeException`. Bien sûr, il est possible de créer nous-mêmes nos exceptions afin de lever nos propres exceptions correspondant à des cas limites fonctionnels ou techniques. Par exemple, imaginons un site d'e-commerce qui affiche une page correspondant au descriptif d'un produit afin de pouvoir le commander. Nous chargeons le produit. Si le produit n'est plus en stock alors il peut être judicieux de lever une exception afin que le site puisse gérer ce cas limite et afficher un message en conséquence. Créons donc notre exception personnalisée : `ProduitNonEnStockException`...

Pour ce faire, il suffit de créer une classe qui dérive de la classe de base `Exception` :

```
public class ProduitNonEnStockException : Exception
{
}
```

Qui pourra être utilisée ainsi :

```
static void Main(string[] args)
{
    try
    {
        ChargerProduit("TV HD");
    }
    catch (ProduitNonEnStockException ex)
```

```
    {
        Console.WriteLine("Erreur : " + ex.Message);
    }
}

public static Produit ChargerProduit(string nomProduit)
{
    Produit produit = new Produit(); // à remplacer par le chargement du produit
    if (produit.Stock <= 0)
        throw new ProduitNonEnStockException();
    return produit;
}
```

Il serait intéressant de pouvoir rendre l'exception plus explicite en modifiant par exemple la propriété message de l'exception. Pour ce faire, il suffit d'utiliser la surcharge du constructeur prenant une chaîne de caractères en paramètre afin de pouvoir mettre à jour la propriété `Message` (qui est en lecture seule) :

```
public class ProduitNonEnStockException : Exception
{
    public ProduitNonEnStockException() : base("Le produit n'est pas en stock")
    {
    }
}
```

Nous pouvons également créer un constructeur qui prend le nom du produit en paramètre afin de rendre le message encore plus précis :

```
public class ProduitNonEnStockException : Exception
{
    public ProduitNonEnStockException(string nomProduit) : base("Le produit " + nomProduit + " n'est pas en stock")
    {
    }
}
```

Que nous pourrons utiliser ainsi :

```
public static Produit ChargerProduit(string nomProduit)
{
    Produit produit = new Produit(); // à remplacer par le chargement du produit
    if (produit.Stock <= 0)
        throw new ProduitNonEnStockException(nomProduit);
    return produit;
}
```

Ce qui donne une belle levée d'exception :

```
Erreur : Le produit TV HD n'est pas en stock
```

À noter que pour construire cette exception personnalisée, nous avons dérivé de la classe de base `Exception`. Il aurait été également possible de dériver de la classe `ApplicationException` pour conserver une hiérarchie cohérente d'exceptions.

Le mot-clé `finally`

Nous avons vu que lorsqu'un code est surveillé dans un bloc `try...catch`, on sortait du bloc `try` si jamais une exception était levée pour atterrir dans le bloc `catch`. Cela veut dire qu'il est impossible de garantir qu'une instruction sera exécutée dans notre code, si jamais une exception nous fait sortir du bloc. C'est là qu'intervient le mot-clé `finally`. Il permet d'indiquer que dans tous les cas, un code doit être exécuté, qu'une exception intervienne ou pas. Par exemple :

```
try
{
    string saisie = Console.ReadLine();
    int valeur = Convert.ToInt32(saisie);
    Console.WriteLine("Vous avez saisi un entier");
}
catch (FormatException)
{
    Console.WriteLine("Vous avez saisi autre chose qu'un entier
        ");
}
finally
{
    Console.WriteLine("Merci d'avoir saisi quelque chose");
}
```

Nous demandons une saisie utilisateur. Nous tentons de convertir cette saisie en entier. Si la conversion fonctionne, nous restons dans le bloc `try` :

```
111
Merci d'avoir saisi quelque chose
```

Si la conversion lève une `FormatException`, le texte affiché dans la console sera différent :

```
abc
Vous avez saisi autre chose qu'un entier
Merci d'avoir saisi quelque chose
```

Dans les deux cas, nous passons obligatoirement dans le bloc `finally` pour afficher le remerciement.

Le bloc `finally` est utile par exemple quand il s'agit de libérer la mémoire, d'enregistrer des données, d'écrire dans un fichier de log, etc.

Il est important de remarquer que quelle que soit la construction, le bloc `finally` est toujours exécuté. Ainsi, même si on relève une exception dans le bloc `catch` :

```
try
{
    Convert.ToInt32("ppp");
}
catch (FormatException)
{
    throw new NotImplementedException();
}
finally
{
    Console.WriteLine("Je suis quand même passé ici");
}
```

nous afficherons toujours notre message...

Même lorsque nous souhaitons sortir d'une méthode avec le mot-clé `return` :

```
static void Main(string[] args)
{
    MaMethode();
}

private static void MaMethode()
{
    try
    {
        Convert.ToInt32("ppp");
    }
    catch (FormatException)
    {
        return;
    }
    finally
    {
        Console.WriteLine("Je suis quand même passé ici");
    }
}
```

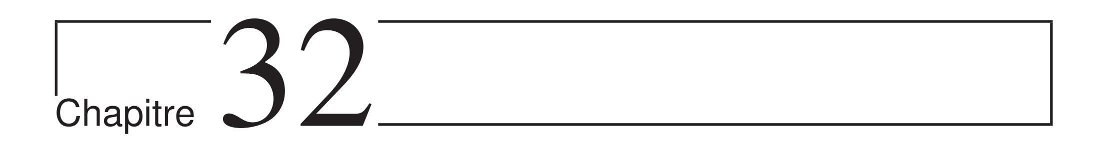
Ici, le bloc `finally` est quand même exécuté.

Dans ce chapitre, nous avons vu comment gérer les cas limites en utilisant le mécanisme des exceptions. Vous avez pu voir qu'il est très pratique de gérer les exceptions grâce à leurs types et qu'il est finalement très facile de créer nos propres exceptions.

Attention : la gestion des exceptions ne doit pas masquer les bugs. Vous ne devez pas entourer tout votre code de blocs `try...catch` et vous féliciter qu'il n'y ait aucun bug. C'est au contraire une opportunité pour les identifier et produire un rapport pour les corriger ultérieurement, par exemple en envoyant un e-mail avec le contenu du bug aux développeurs.

En résumé

- Les exceptions permettent de gérer les cas limites d'une méthode.
- On utilise le bloc `try...catch` pour encapsuler un bout de code à surveiller.
- Il est possible de créer des exceptions personnalisées en dérivant de la classe de base `Exception`.
- On peut lever à tout moment une exception grâce au mot-clé `throw`.
- Les exceptions ne doivent pas servir à masquer les bugs.



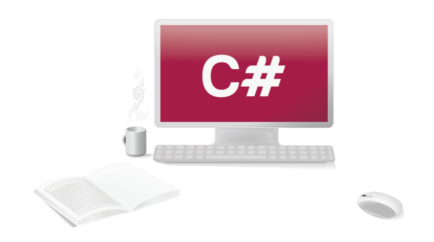
Chapitre 32

TP : événements et météo

Difficulté : 

Bienvenue dans le dernier TP de cette partie. Tenez bon, après on change de domaine pour aborder d'autres notions. Dans ce TP, nous allons pratiquer les événements. Le but est de savoir en créer un et de pouvoir s'y abonner pour être notifié d'une information.

Vous êtes prêts ? Alors c'est parti !



Instructions pour réaliser le TP

Nous allons réaliser ici un mini simulateur de météo qui sera utilisé par un statisticien afin d'en faire des statistiques (logique !). Bon, c'est le contexte, mais c'est juste un exemple. N'espérez pas non plus réaliser un vrai simulateur météo dans ce TP !

Bref : nous devons créer un simulateur de météo. Lorsque celui-ci est démarré, il génère autant de nombres aléatoires que demandé, des nombres entre 0 et 100. Si le nombre aléatoire est inférieur à 5, alors ça veut dire que le temps est au soleil. S'il est supérieur ou égal à 5 et inférieur à 50, alors nous aurons des nuages. S'il est supérieur ou égal à 50 et inférieur à 90, alors nous aurons de la pluie. Sinon, nous aurons de l'orage. Un événement sera levé à chaque changement de temps. Le but de notre statisticien est de s'abonner aux événements du simulateur météo afin de compter le nombre de fois où il a fait soleil et le nombre de fois où le temps a changé. Il affichera ensuite son rapport en indiquant ces deux résultats et le pourcentage de fois où il a fait soleil (je veux bien que ce pourcentage soit un entier).

C'est tout pour l'énoncé ! Maintenant, vous avez assez de connaissances pour que je ne détaille pas plus. Bon courage !

Correction

Allez, c'est parti pour la correction. Tout d'abord, nous devons créer notre simulateur météo. Il sera représenté par une classe :

```
public class SimulateurMeteo
{
}
```

Nous aurons également besoin de quelque chose pour représenter le temps : soleil, nuage, pluie et orage. Une énumération semble appropriée :

```
public enum Temps
{
    Soleil,
    Nuage,
    Pluie,
    Orage
}
```

Enfin, nous aurons notre statisticien :

```
public class Statisticien
{
}
```

Commençons par le simulateur de météo. Nous aurons besoin de plusieurs variables membres afin de stocker notre générateur de nombres aléatoires, le dernier temps qu'il a fait et le nombre de répétitions. Le nombre de répétitions pourra être un paramètre du constructeur :

```
public class SimulateurMeteo
{
    private Temps? ancienTemps;
    private int nombreDeRepetitions;
    private Random random;

    public SimulateurMeteo(int nombre)
    {
        random = new Random();
        ancienTemps = null;
        nombreDeRepetitions = nombre;
    }
}
```

Étant donné que nous allons déterminer plusieurs nombres aléatoires, il est pertinent de ne pas réallouer à chaque fois le générateur de nombre aléatoire. C'est pour cela qu'on le crée une unique fois dans le constructeur de la classe. Créons maintenant une méthode permettant de démarrer le simulateur et codons les règles métier du simulateur :

```
public class SimulateurMeteo
{
    //[...] Code supprimé pour plus de clarté [...]

    public void Demarrer()
    {
        for (int i = 0; i < nombreDeRepetitions; i++)
        {
            int valeur = random.Next(0, 100);
            if (valeur < 5)
                GererTemps(Temps.Soleil);
            else
            {
                if (valeur < 50)
                    GererTemps(Temps.Nuage);
                else
                {
                    if (valeur < 90)
                        GererTemps(Temps.Pluie);
                    else
                        GererTemps(Temps.Orage);
                }
            }
        }
    }
}
```

C'est très simple, on boucle sur le nombre de répétitions. Un nombre aléatoire est déterminé à chaque itération. La méthode `GererTemps` prend en paramètre le temps déterminé à partir du nombre aléatoire. C'est cette méthode `GererTemps` qui aura pour but de lever un événement quand le temps change :

```
public class SimulateurMeteo
{
    public delegate void IlFaitBeauDelegate(Temps temps);
    public event IlFaitBeauDelegate QuandLeTempsChange;

    // [...] Code supprimé pour plus de clarté [...]

    private void GererTemps(Temps temps)
    {
        if (ancienTemps.HasValue && ancienTemps.Value != temps
            && QuandLeTempsChange != null)
             QuandLeTempsChange(temps);
        ancienTemps = temps;
    }
}
```

Ici, j'ai choisi de créer un seul événement quand le temps change et de lui indiquer le temps qu'il fait en paramètre. Nous avons donc besoin d'un délégué qui prend un `Temps` en paramètre et qui ne renvoie rien - c'est d'ailleurs souvent le cas des événements. Puis nous avons besoin d'un événement du type du délégué. Ensuite, si le temps a changé et que quelqu'un s'est abonné à l'événement, alors nous levons l'événement.

Il ne reste plus qu'à remplir notre classe `Statisticien`. Cette classe a besoin de travailler sur une instance de la classe `SimulateurMeteo` ; nous pouvons donc lui en passer une dans les paramètres du constructeur. Nous utiliserons également des variables membres privées permettant de stocker ses analyses :

```
public class Statisticien
{
    private SimulateurMeteo simulateurMeteo;
    private int nombreDeFoisOuLeTempsAChange;
    private int nombreDeFoisOuIlAFaitSoleil;

    public Statisticien(SimulateurMeteo simulateur)
    {
        simulateurMeteo = simulateur;
        nombreDeFoisOuLeTempsAChange = 0;
        nombreDeFoisOuIlAFaitSoleil = 0;
    }

    public void DemarrerAnalyse()
    {
        simulateurMeteo.QuandLeTempsChange +=
            simulateurMeteo_QuandLeTempsChange;
        simulateurMeteo.Demarrer();
    }

    public void AfficherRapport()
    {
        Console.WriteLine("Nombre de fois où le temps a changé
            : " + nombreDeFoisOuLeTempsAChange);
```

```
        Console.WriteLine("Nombre de fois où il a fait soleil : " + nombreDeFoisOuIlAFaitSoleil);
        Console.WriteLine("Pourcentage de beau temps : " + nombreDeFoisOuIlAFaitSoleil * 100 / nombreDeFoisOuLeTempsAChange + " %");
    }

    private void simulateurMeteo_QuandLeTempsChange(Temps temps)
    {
        if (temps == Temps.Soleil)
            nombreDeFoisOuIlAFaitSoleil++;
        nombreDeFoisOuLeTempsAChange++;
    }
}
```

Notons que dans la méthode `DemarrerAnalyse`, nous nous abonnons à l'événement de changement de temps. La méthode qui est appelée lors de la notification est très simple, elle incrémente les compteurs. Enfin, l'affichage du rapport est trivial. Ici, comme nous n'avons que des entiers, la division produira un entier également. Il ne reste plus qu'à faire fonctionner nos objets :

```
class Program
{
    static void Main(string[] args)
    {
        SimulateurMeteo simulateurMeteo = new SimulateurMeteo(1000);
        Statisticien statisticien = new Statisticien(simulateurMeteo);
        statisticien.DemarrerAnalyse();
        statisticien.AfficherRapport();
    }
}
```

Ici, je travaille sur 1000 répétitions. Voici le résultat que j'obtiens à l'exécution de l'application :

```
Nombre de fois où le temps a changé : 646
Nombre de fois où il a fait soleil : 55
Pourcentage de beau temps : 8 %
```

Évidemment, vu que nous travaillons avec des nombres aléatoires, chacun aura un résultat différent.

Et voilà, c'est terminé pour ce TP. Notre application est fonctionnelle !

Terminé ? hmm... pas tout à fait, allons un peu plus loin.

Aller plus loin

En l'état, ce code est fonctionnel. C'est parfait. Mais que se passe-t-il si nous démarrons plusieurs fois l'analyse ? Nous pouvons essayer :

```
static void Main(string[] args)
{
    SimulateurMeteo simulateurMeteo = new SimulateurMeteo(1000)
        ;
    Statisticien statisticien = new Statisticien(
        simulateurMeteo);
    statisticien.DemarrerAnalyse();
    statisticien.AfficherRapport();

    statisticien.DemarrerAnalyse();
    statisticien.AfficherRapport();

    statisticien.DemarrerAnalyse();
    statisticien.AfficherRapport();
}
```

Ce qui affichera :

```
Nombre de fois où le temps a changé : 635
Nombre de fois où il a fait soleil : 47
Pourcentage de beau temps : 7 %
Nombre de fois où le temps a changé : 1917
Nombre de fois où il a fait soleil : 163
Pourcentage de beau temps : 8 %
Nombre de fois où le temps a changé : 3879
Nombre de fois où il a fait soleil : 322
Pourcentage de beau temps : 8 %
```

Les valeurs augmentent... alors qu'elles ne devraient pas. Eh oui, nous ne réinitialisons pas les entiers permettant de stocker les statistiques. Ce n'est pas forcément un bug. Ici, comme je n'avais rien dit dans l'énoncé, il peut paraître pertinent de continuer à incrémenter ces valeurs, comme ça, je peux travailler sur une moyenne. Bon, disons que nous souhaitons les réinitialiser à chaque fois ; il suffit alors de remettre à zéro les compteurs dans la méthode :

```
public void DemarrerAnalyse()
{
    nombreDeFoisOuLeTempsAChange = 0;
    nombreDeFoisOuIlAFaitSoleil = 0;
    simulateurMeteo.QuandLeTempsChange +=
        simulateurMeteo_QuandLeTempsChange;
    simulateurMeteo.Demarrer();
}
```

Cependant, les compteurs augmentent toujours, moins vite, mais quand même ! Je suis sûr que vous l'aviez deviné et que vous n'avez pas fait l'erreur. En fait, cela vient

de l'abonnement à l'événement. Chaque fois que nous démarrons l'analyse, nous nous réabonnons à l'événement. Comme l'événement est multidiffusion, nous rajoutons en fait à chaque fois un appel à la méthode avec le +=. Ce qui veut dire qu'à la deuxième fois, nous appellerons la méthode deux fois, ce qui fera doubler les résultats. À la troisième fois, ils triplent... Nous avons donc une erreur. Soit il faut s'abonner une seule fois à l'événement, soit il faut se désabonner à la fin de l'analyse, quand ce n'est plus utile de recevoir l'événement. C'est cette dernière méthode que je vais vous montrer. Il suffit de faire :

```
public void DemarrerAnalyse()
{
    nombreDeFoisOuLeTempsAChange = 0;
    nombreDeFoisOuIlAFaitSoleil = 0;
    simulateurMeteo.QuandLeTempsChange +=
        simulateurMeteo_QuandLeTempsChange;
    simulateurMeteo.Demarrer();
    simulateurMeteo.QuandLeTempsChange -=
        simulateurMeteo_QuandLeTempsChange;
}
```

On utilise l'opérateur -= pour enlever la méthode du délégué.



D'une manière générale, il est bienvenu de se désabonner d'un événement lorsque l'on sait qu'on ne va plus s'en servir.

Cela permet d'éviter d'encombrer la mémoire qui ne saurait pas forcément se libérer toute seule. Je n'en dis pas plus car ceci est un concept avancé de gestion de mémoire. Gardez seulement à l'esprit que si on a l'opportunité de se désabonner d'un événement, il faut le faire.

Enfin, nous pouvons simplifier notre code en ne créant pas notre délégué. Effectivement, dans la mesure où celui-ci possède un seul paramètre, il est possible de le remplacer par le délégué `Action<T>`. Il faut juste supprimer la déclaration du délégué et remplacer la déclaration de l'événement par :

```
public class SimulateurMeteo
{
    public event Action<Temps> QuandLeTempsChange;
}
```

Vous pouvez télécharger tous les codes sources de cet exercice grâce au code web suivant :

▷ Copier ce code
Code web : 179137

Quatrième partie

C# avancé

Chapitre 33

Créer un projet bibliothèques de classes

Difficulté : 

Pour l'instant, nous n'avons créé que des projets de type application console. Il existe plein d'autres modèles de projet. Un des plus utiles est le projet permettant de créer des bibliothèques de classes. Il s'agit d'un projet qui va permettre de contenir des classes que nous pourrons utiliser dans des applications. Exactement comme la bibliothèque de classes du framework .NET qui nous permet d'utiliser la classe `Math` ou les exceptions, ou plein d'autres choses. . . Ce type de projet permet donc de créer une assembly sous la forme d'un fichier avec l'extension `.dll`. Ces assemblys sont donc des bibliothèques qui seront utilisables par n'importe quelle application utilisant un langage compatible avec le framework .NET.



Pourquoi créer une bibliothèque de classes ?

Globalement pour deux raisons que nous allons détailler :

- Réutilisabilité.
- Architecture.

Réutilisabilité :

Comme indiqué en introduction, le projet de type bibliothèque de classes permet d'obtenir des assemblys avec l'extension `.dll`. Nous pouvons y mettre tout le code C# que nous voulons, notamment des classes qui auront un intérêt à être utilisées à plusieurs endroits ou partagées par plusieurs applications. C'est le cas des assemblys du framework .NET. Elles possèdent plein de code très utile que nous aurons avantage à utiliser pour créer nos applications.

De la même façon, nous allons pouvoir créer des classes qui pourront être réutilisées à plusieurs endroits. Comme notre classe permettant de gérer les listes chaînées. N'importe quel programme qui utilise des listes chaînées aura intérêt à ne pas tout réinventer et à simplement réutiliser ce code prêt à l'emploi. Il suffira pour ce faire de réutiliser cette classe en référençant l'assembly, comme nous l'avons déjà vu.

Architecture :

L'autre avantage dans la création de bibliothèques de classes est de pouvoir architecturer son application de manière à faciliter sa mise en place, sa maintenabilité et l'évolutivité du code. L'architecture informatique c'est comme l'architecture d'une maison. Si nous mettons la douche au même endroit que le compteur électrique, ou que la machine à laver est juste à côté de la chambre à coucher, cela peut poser des problèmes. De même, que penser d'un architecte qui place les toilettes à côté du lit ? Une maison mal architecturée est une maison où il ne fait pas bon vivre. De même, une application mal architecturée est une application dont il ne fait pas bon faire la maintenance applicative !

Architecturer son application est important surtout si l'application est grosse. Par exemple, une bonne pratique dans une application informatique est la décomposition en couches. Nous n'allons pas faire ici un cours d'architecture, mais le but est de séparer les composantes de l'application. Un grand nombre d'applications de gestion est composé d'une couche de présentation, d'une couche métier et d'une couche d'accès aux données, les couches communiquant entre elles. Le but est de limiter les modifications d'une couche lors de la modification d'une autre. Si toutes les couches étaient mélangées alors chaque modification impliquerait une série de modifications en chaîne. On peut faire une analogie avec un plat de lasagnes et un plat de spaghettis. Il est difficile de toucher à un spaghetti sans toucher les autres. Cependant, il pourrait être envisageable de toucher à la couche du dessus du plat de lasagnes pour rajouter un peu de fromage sans perturber ce qu'il y a dessous. Miam !

Il est donc intéressant d'avoir une bibliothèque de classes qui gère l'accès aux données,

une autre qui gère les règles métier et une autre qui s'occupe d'afficher le tout. Je m'arrête là sur l'architecture, vous aurez l'occasion de vous y confronter bien assez tôt !

Créer un projet de bibliothèque de classe

Pour créer une bibliothèque de classes, on utilise l'assistant de création de nouveau projet (`Fichier` > `Nouveau` > `Nouveau projet`), comme on l'a fait pour une application console sauf qu'ici, on utilisera le modèle `Bibliothèque de classes` (voir la figure 33.1). Ne le faisons pas encore.

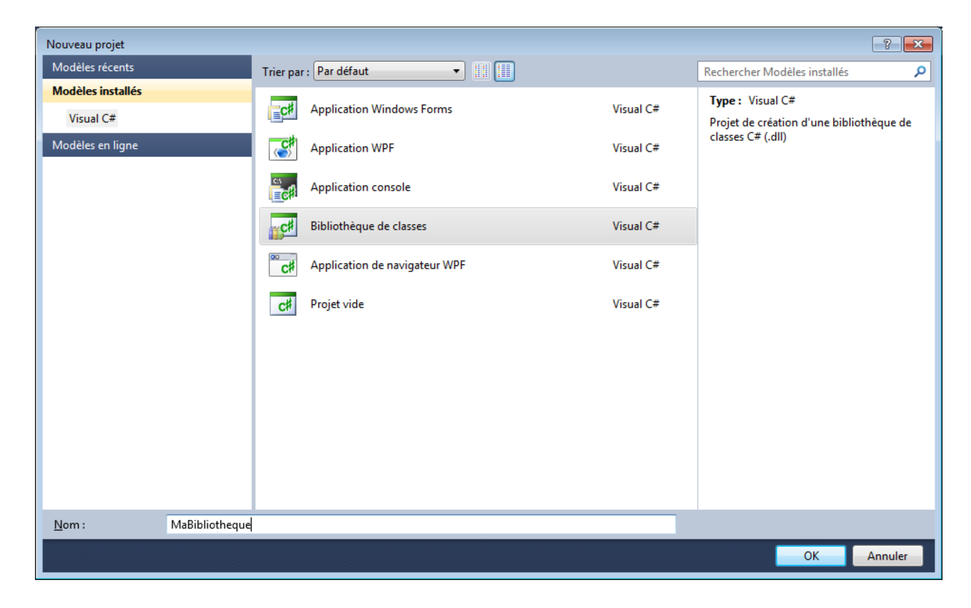


FIGURE 33.1 – Choix d'un projet de type Bibliothèque de classes

Si nous faisons cela, Visual C# Express va nous créer une nouvelle solution contenant le projet de bibliothèques de classes. C'est possible sauf qu'en général, on ajoute une bibliothèque de classes pour qu'elle serve dans le cadre d'une application. Cela veut dire que nous pouvons ajouter ce nouveau projet à notre solution actuelle. Ainsi, celle-ci contiendra deux projets :

- **L'application console**, qui sera exécutable par le système d'exploitation.
- **La bibliothèque de classes**, qui sera utilisée par l'application.

Pour ce faire, on peut faire un clic droit sur notre solution et faire `Ajouter` > `Nouveau Projet`, comme indiqué à la figure 33.2.

Ici, on choisit le projet de type bibliothèque de classes, sans oublier de lui donner un nom, par exemple `MaBibliotheque` (voir figure 33.3). Visual C# Express nous crée notre projet et l'ajoute à la solution, comme on peut le voir dans l'explorateur de solutions (voir figure 33.4).

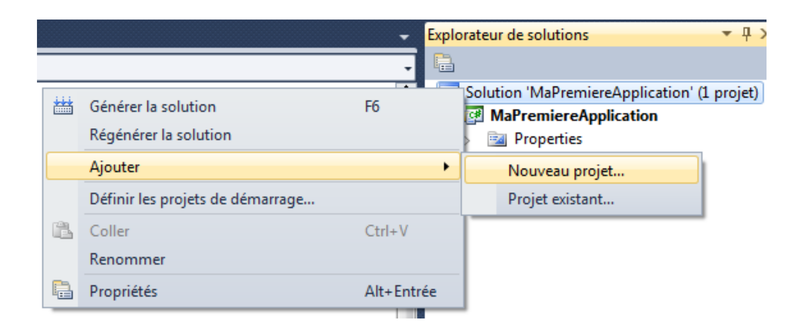


FIGURE 33.2 – Ajout d'un nouveau projet à la solution

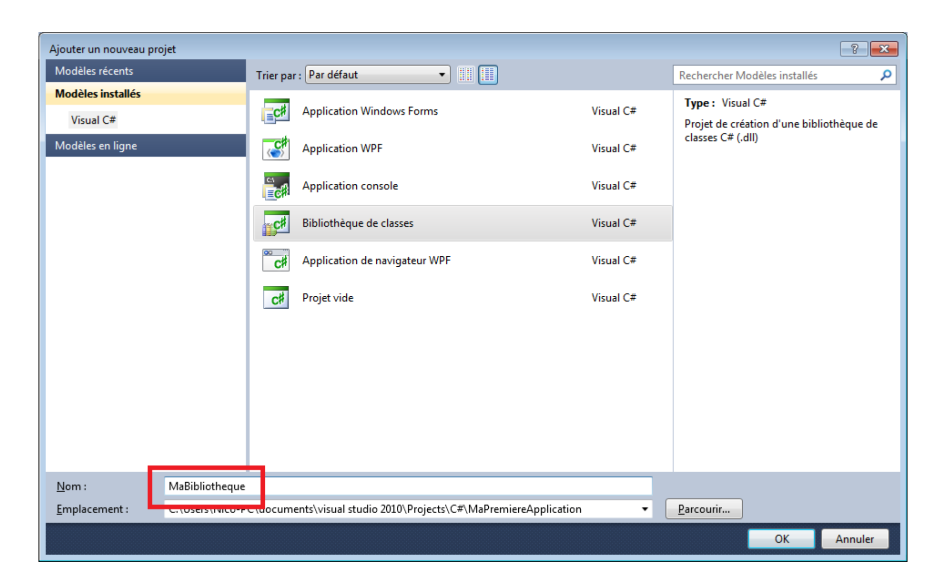


FIGURE 33.3 – Création d'un projet Bibliothèque de classes appelé `MaBibliotheque`

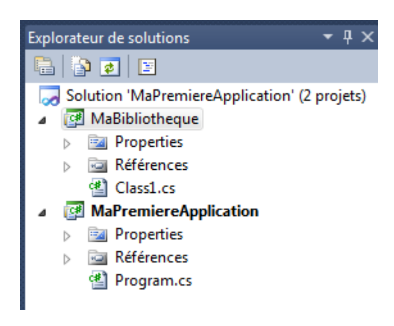


FIGURE 33.4 – L'explorateur de solution affiche les deux projets contenus dans la solution

Il le génère avec un fichier `Class1.cs` que nous pouvons supprimer.

Remarquez que l'application console est en gras. Cela veut dire que c'est ce projet que Visual C# Express va tenter de démarrer lorsque nous utiliserons les touches `Ctrl` + `F5`. Si ce n'est pas le cas, il peut tenter de démarrer la bibliothèque, ce qui n'a pas de sens vu qu'elle n'a pas de méthode `Main()`. Dans ce cas, vous pourrez le changer en cliquant droit sur le projet console et en choisissant `Définir comme projet de démarrage`.

Dans ce projet, nous pourrons désormais créer nos classes en ajoutant de nouvelles classes au projet, comme on l'a déjà fait avec l'application console.

Ajoutons par exemple la classe suivante :

```
namespace MaBibliotheque
{
    public class Bonjour
    {
        public void DireBonjour()
        {
            Console.WriteLine("Bonjour depuis la bibliothèque
                MaBibliotheque");
        }
    }
}
```

Propriétés de la bibliothèque de classe

Ouvrons la fenêtre des propriétés de notre projet, en faisant un clic droit sur le projet, puis `Propriétés` (voir la figure 33.5).

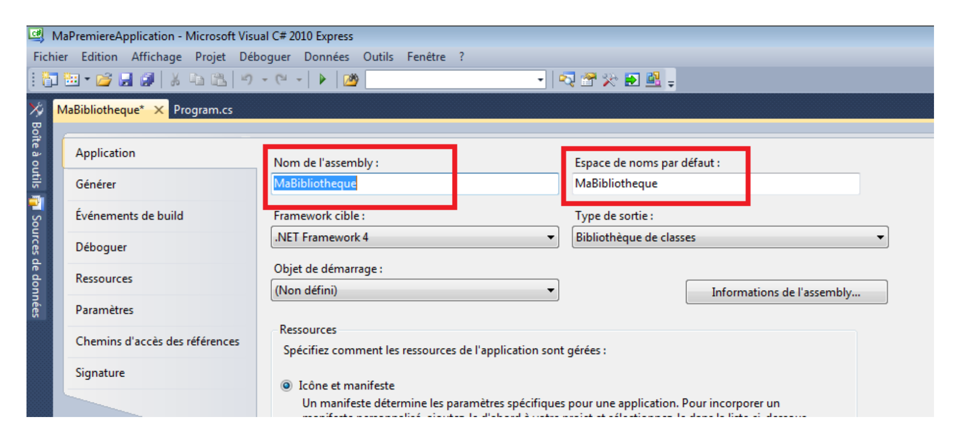


FIGURE 33.5 – Le nom de l'assembly et l'espace de noms par défaut dans les propriétés du projet

Nous pouvons voir différentes informations et notamment dans l'onglet `Application`

que le projet possède un nom d'assembly, qui correspond ici au nom du projet ainsi qu'un espace de noms par défaut, qui correspond également au nom du projet. Le nom de l'assembly servira à identifier l'assembly, alors que l'espace de noms par défaut sera celui donné lorsque nous ajouterons une nouvelle classe (ou autre) à notre projet. Cet espace de noms pourra être changé, mais en général on garde un nom cohérent avec les arborescences de notre projet.

Notons au passage qu'il y a une option permettant d'indiquer la version du framework .NET que nous souhaitons utiliser. Ici, nous gardons la version 4.

Générer et utiliser une bibliothèque de classe

Nous pouvons alors compiler cette bibliothèque de classes, soit individuellement, soit en compilant tout le projet.

Rendons-nous dans le répertoire où nous avons sauvegardé notre bibliothèque. Nous voyons dans le répertoire de sortie (chez moi : `C:\Users\Nico-PC\Documents\Visual Studio 2010\Projects\C#\MaPremiereApplication\MaBibliotheque\bin\Release`) qu'il y a un fichier du nom de notre projet dont l'extension est `.dll`. C'est notre bibliothèque de classes (même s'il n'y a qu'une classe dedans !). Elle possède le même nom que celui que nous avons vu dans les propriétés du projet.

J'en profite pour vous faire remarquer qu'à son côté, il y a également un fichier du même nom avec l'extension `.pdb`. Je peux enfin vous révéler de quoi il s'agit. Ce fichier contient les informations de débogages, utiles lorsque nous déboguons notre application. Sans ce fichier, il est impossible de déboguer à l'intérieur du code de notre classe.

Revenons à l'assembly générée. C'est cette `dll` qu'il faudra référencer dans notre projet, comme nous l'avons vu au chapitre sur le référencement d'assemblys. Si vous avez un doute, n'hésitez pas à retourner le consulter. Ainsi, nous serons en mesure d'utiliser la classe de notre bibliothèque. Rappelez-vous, pour référencer une assembly, nous faisons un clic droit sur les références du projet et sélectionnons `Ajouter une référence`. Nous pourrons référencer notre bibliothèque de deux manières ; soit en utilisant l'onglet `Parcourir`, qui va nous permettre de pointer directement le fichier `dll` contenant nos classes (voir figure 33.6) ; soit en référençant directement le projet de bibliothèque de classes qui se trouve dans notre solution en utilisant l'onglet `Projet` (voir figure 33.7).

C'est ce choix qui doit être privilégié lorsque notre solution contient les projets à référencer. Généralement, vos bibliothèques vont évoluer en même temps que votre programme, donc il est judicieux de les avoir dans la même solution que l'application. Nous pourrons donc faire des références projet. Si cependant les bibliothèques sont stables et ne sont pas amenées à évoluer, alors vous pourrez les référencer directement à partir des `dll` ; vous y gagnerez en temps de compilation. Utilisons désormais notre classe avec le code suivant :

```
MaBibliotheque.Bonjour bonjour = new MaBibliotheque.Bonjour();
bonjour.DireBonjour();
```

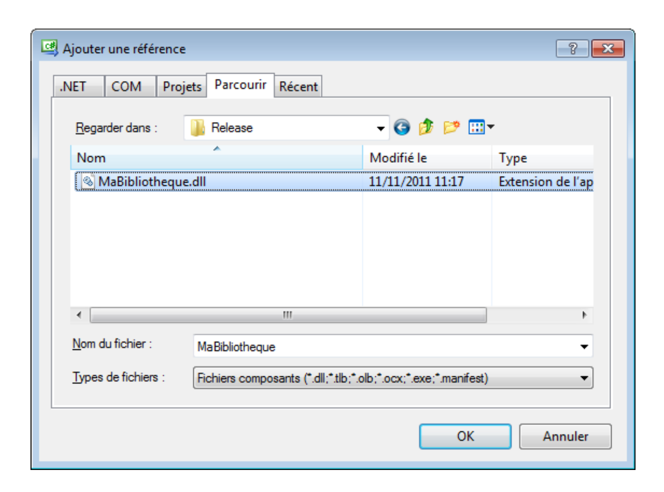


FIGURE 33.6 – Référencement direct de l'assembly

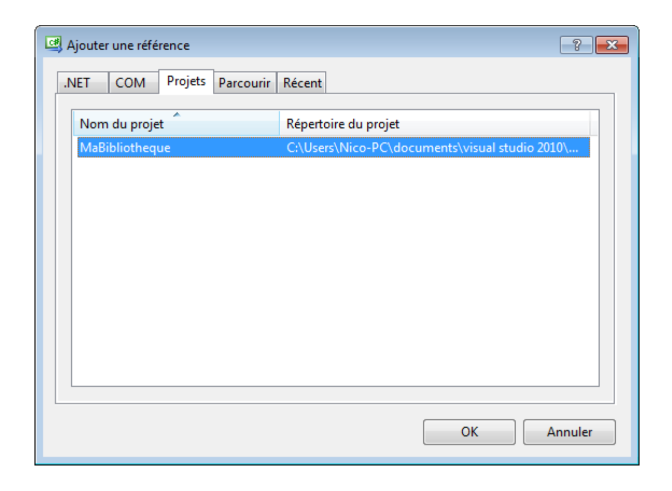


FIGURE 33.7 – Référencement du projet

Vous pouvez aussi, bien sûr, inclure l'espace de noms `MaBibliotheque` pour éviter d'avoir à préfixer la classe. En général, l'espace de noms d'une bibliothèque est différent de celui de l'application. Notez que pour qu'une classe, comme la classe `Bonjour`, puisse être utilisée par une application référençant son assembly, elle doit être déclarée en `public` afin qu'elle soit visible par tout le monde.

Le mot-clé `internal`

Nous avons déjà vu ce mot-clé lorsque nous avons parlé de visibilité avec notamment les autres mots-clés `public`, `private` et `protected`.

C'est avec les assemblys que le mot-clé `internal` prend tout son sens. Il permet d'indiquer qu'une classe, méthode ou propriété ne sera accessible qu'à l'intérieur d'une assembly. Cela permet par exemple qu'une classe soit utilisable par toutes les autres classes de cette assembly mais qu'elle ne puisse pas être utilisée par une application référençant l'assembly.

Créons par exemple dans notre bibliothèque de classes les trois classes suivantes :

```
public class Client
{
    private string login;
    public string Login
    {
        get { return login; }
    }

    private string motDePasse;
    public string MotDePasse
    {
        get { return motDePasse.Crypte(); }
    }

    public Client(string loginClient, string motDePasseClient)
    {
        login = loginClient;
        motDePasse = motDePasseClient;
    }
}

public static class Generateur
{
    public static string ObtenirIdentifiantUnique()
    {
        Random r = new Random();
        string chaine = string.Empty;
        for (int i = 0; i < 10; i++)
        {
            chaine += r.Next(0, 100);
```

```
        }
        return chaine.Crypte();
    }
}

internal static class Encodage
{
    internal static string Crypte(this string chaine)
    {
        return Convert.ToBase64String(Encoding.Default.GetBytes
            (chaine));
    }

    internal static string Decrypte(this string chaine)
    {
        return Encoding.Default.GetString(Convert.
            FromBase64String(chaine));
    }
}
```

Vous pouvez copier ce code en utilisant le code web suivant :

▷ Copier ce code
Code web : 888787

Les deux premières sont des classes publiques qui peuvent être utilisées depuis n'importe où, par exemple depuis le programme principal :

```
class Program
{
    static void Main(string[] args)
    {
        Client client = new Client("Nico", "12345");
        Console.WriteLine(client.MotDePasse);

        Console.WriteLine(Generateur.ObtenirIdentifiantUnique()
            );
    }
}
```

Par contre, la classe `Encodage` n'est accessible que pour les deux autres classes, car elles sont dans la même assembly. Cela veut dire que si nous tentons de l'utiliser depuis notre méthode `Main()`, nous aurons des erreurs de compilation :

```
static void Main(string[] args)
{
    string chaine = "12345".Crypte();

    Encodage.Crypte("12345");
}
```

Cet exemple permet d'illustrer l'intérêt, qui n'est pas toujours évident, du mot-clé `internal`.

À noter qu'il existe enfin le mot-clé `protected internal` qui permet d'indiquer que des éléments sont accessibles à un niveau `internal` pour les classes d'une même assembly mais `protected` pour les autres assemblys, ce qui permet d'appliquer les principes de substitutions ou d'héritage.

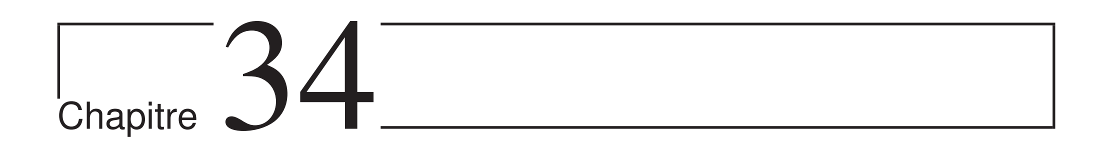
Voilà, vous avez vu comment créer une bibliothèque de classes ! N'hésitez pas à créer ce genre de projet afin d'y mettre toutes les classes qui peuvent être utilisées par plusieurs applications. Comme ça, il suffit d'une simple référence pour accéder au code qui y est contenu. Vous vous en servirez également pour mieux architecturer vos applications, le code s'en trouvera plus clair et plus maintenable.



La traduction en anglais de bibliothèque est « *library* », vous verrez souvent ce mot-là sur internet. Vous le verrez également mal francisé en « librairie », ce qui est évidemment une erreur !

En résumé

- Un projet de bibliothèque de classes permet de regrouper des classes pouvant être utilisées entre plusieurs applications.
- Les bibliothèques de classes génèrent des assemblys avec l'extension `.dll`.
- Elles permettent également de mieux architecturer un projet.



Chapitre 34

Plus loin avec le C# et .NET

Difficulté : 

Maintenant que nous en savons plus, nous allons pouvoir aborder quelques notions qui me paraissent importantes et que nous n'avons pas encore traitées. Ce sera l'occasion d'approfondir nos connaissances et de comprendre un peu mieux certains points qui auraient pu vous paraître douteux.

Nous allons voir des instructions C# supplémentaires qui vont nous permettre de compléter nos connaissances en POO. Nous en profiterons également pour détailler comment le framework .NET gère les types en mémoire. Vous en saurez plus sur le formatage des chaînes et aurez un aperçu du côté obscur du framework .NET, la réflexion !

Bref, tout plein de nouvelles cordes à nos arcs nous permettant d'être de plus en plus efficace avec le C#. Ces nouveaux concepts vous serviront sans doute moins souvent, mais sont importants à connaître.



Empêcher une classe de pouvoir être héritée

Parmi ces nouveaux concepts, nous allons voir comment il est possible de faire en sorte qu'une classe ne puisse pas être héritée. Rappelez-vous, à un moment, j'ai dit qu'on ne pouvait pas créer une classe qui spécialise la classe `String`.



C'est quoi ce mystère ? Nous, quand nous créons une classe, n'importe qui peut en hériter. Pourquoi pas la classe `String` ?

C'est parce que je vous avais caché le mot-clé `sealed`. Il permet d'empêcher la classe d'être héritée. Tout simplement !

Pourquoi vouloir empêcher d'étendre une classe en la dérivant ? Pour plusieurs raisons, mais généralement nous allons recourir à ce mot-clé lorsqu'une classe est trop spécifique à une méthode ou à une bibliothèque et que l'on souhaite empêcher quelqu'un de pouvoir obtenir du code instable en étendant son fonctionnement. C'est typiquement le cas pour la classe `String`. Il y a beaucoup de classes dans le framework .NET qui sont marquées comme `sealed` afin d'éviter que l'on puisse faire n'importe quoi.

Il faut par contre faire attention car l'emploi de ce mot-clé restreint énormément la façon dont on pourrait employer cette classe. Pour déclarer une classe `sealed`[1] il suffit de faire précéder le mot-clé `class` du mot-clé `sealed` :

```
public sealed class ClasseImpossibleADeriver
{
}
```

Ainsi, toute tentative d'héritage :

```
public class MaClasse : ClasseImpossibleADeriver
{
}
```

se soldera par une erreur de compilation déshonorante :

```
impossible de dériver du type sealed 'MaPremiereApplication.
   ClasseImpossibleADeriver'
```

Le mot-clé `sealed` fonctionne également pour les méthodes, dans ce cas, cela permet d'empêcher la substitution d'une méthode. Reprenons notre exemple du chien muet :

```
public class Chien
{
    public virtual void Aboyer()
    {
        Console.WriteLine("Wouf");
    }
}
```

1. Ce qui signifie « scellée », en anglais.

```

public class ChienMuet : Chien
{
    public sealed override void Aboyer()
    {
        Console.WriteLine("...");
    }
}
```

Ici, nous marquons la méthode comme `sealed`, ce qui fait qu'une classe qui dérive de `ChienMuet` ne sera pas capable de remplacer la méthode permettant d'aboyer. Le code suivant :

```
public class ChienAvecSyntheseVocale : ChienMuet
{
    public override void Aboyer()
    {
        Console.WriteLine("Bwarf");
    }
}
```

entraînera l'erreur de compilation :

```
'MaPremiereApplication.ChienAvecSyntheseVocale.Aboyer()' : ne
   peut pas se substituer à un membre hérité '
   MaPremiereApplication.ChienMuet.Aboyer()', car il est sealed
```

Voilà pour ce mot-clé, à utiliser en connaissance de cause !

Précisions sur les types et gestion mémoire

Nous avons vu beaucoup de types différents au cours des chapitres précédents. Nous avons vu les structures, les classes, les énumérations, les délégués, etc. Certains sont des types valeur et d'autres des types référence ; ils sont tous des objets.

Voici à la figure 34.1 un petit schéma récapitulatif des types que nous avons déjà vus.

Nous avons dit brièvement qu'ils étaient gérés différemment par le framework .NET et que les variables de type valeur contenaient directement la valeur, alors que les variables de type référence possèdent un lien vers un objet en mémoire. Ce qui se passe en fait quand nous déclarons une variable de type valeur, c'est que nous créons directement la valeur en mémoire dans ce qu'on appelle « la pile ». C'est une zone mémoire (limitée) où il est très rapide de mettre et de chercher des valeurs. De plus, cette mémoire n'a pas besoin d'être libérée par le ramasse-miettes (que nous verrons un peu plus loin). C'est pour cela que pour des raisons de performances on peut être amené à choisir les types valeur. Par exemple lorsque nous faisons ce code :

```
int age = 10;
double pourcentage = 5.5;
```

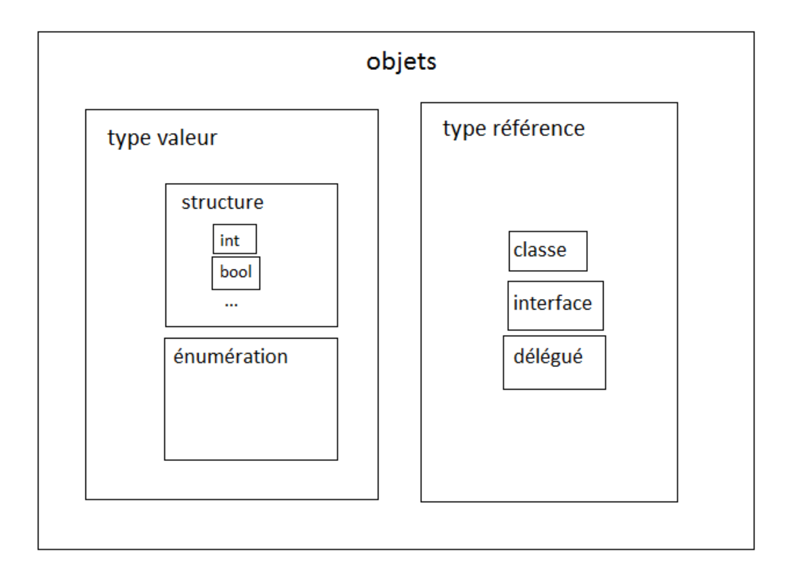


FIGURE 34.1 – Récapitulatif des types

Voici ce qu'il se passe : la variable `age` est une adresse mémoire contenant la valeur 10. C'est la même chose pour la variable `pourcentage` sauf que l'emplacement mémoire est plus important car on peut stocker plus de choses dans un `double` que dans un `int`. Je vous renvoie à la figure 34.2 pour une explication en images ! En ce qui concerne les types référence, lorsque nous instancions par exemple une classe, le C# instancie la variable `maVoiture` dans la pile et lui met dedans une référence vers le vrai objet qui lui est alloué sur une zone mémoire de capacité plus importante, mais à accès plus lent, que l'on appelle « le tas ». Comme il est géré par le framework .NET, on l'appelle « le tas managé ».

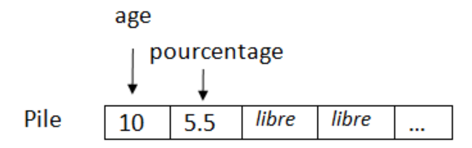


FIGURE 34.2 – Les types valeur sont stockés sur la pile

Si nous avons le code suivant :

```
public class Voiture
{
    public int Vitesse { get; set; }
    public string Marque { get; set; }
}

Voiture maVoiture = new Voiture { Vitesse = 10, Marque = "
```

```
    Peugeot" };
```

nous aurons en mémoire le processus que vous pouvez observer à la figure 34.3.

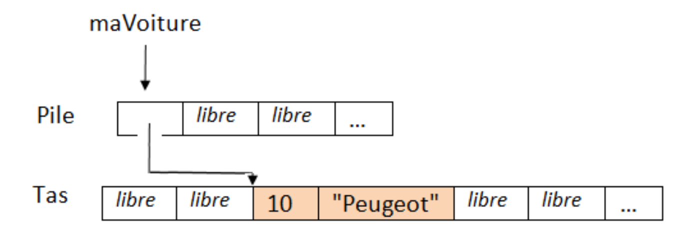


FIGURE 34.3 – Les types référence ont leur référence allouée sur la pile et leur objet alloué sur le tas managé

À noter que lorsqu'une variable sort de sa portée et qu'elle n'est plus utilisable par qui que ce soit, alors la case mémoire sur la pile est marquée comme de nouveau libre. Voilà grosso modo comment est gérée la mémoire.

Il manque encore un élément fondamental du mécanisme de gestion mémoire : **le ramasse-miettes**. Nous en avons déjà parlé brièvement, le ramasse-miettes sert à libérer la mémoire qui n'est plus utilisée dans le tas managé. Il y aurait de quoi écrire un gros tutoriel rien que sur son fonctionnement, aussi nous allons expliquer rapidement comment il fonctionne sans trop rentrer dans les détails.



Le ramasse-miettes est souvent dénommé par sa traduction anglaise, le *garbage collector*.

Pourquoi libérer la mémoire ? On a vu que quand une variable sort de portée, alors la case mémoire sur la pile est marquée comme à nouveau libre. C'est très bien avec les types valeur qui sont uniquement sur la pile. Mais avec les types référence ? Que donne le code suivant lorsque nous quittons la méthode `CreerVoiture` ?

```
public class Voiture
{
    public int Vitesse { get; set; }
    public string Marque { get; set; }
}

static void Main(string[] args)
{
    CreerVoiture();
}

public static void CreerVoiture()
{
```

```
    Voiture maVoiture = new Voiture { Vitesse = 10, Marque = "
        Peugeot" };
}
```

Nous avons dit que la mémoire sur la pile redevenait libre. La référence vers l'objet est donc cassée. Cependant, la mémoire sur le tas est toujours occupée (voir figure 34.4).

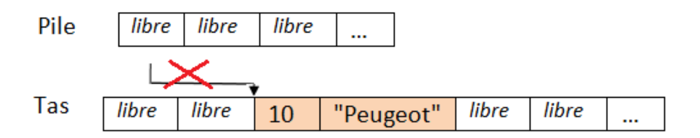


FIGURE 34.4 – L'objet est toujours présent sur le tas, mais n'est plus référencé

Cela veut dire que notre objet est toujours en mémoire sauf que la variable `maVoiture` n'existe plus et donc, ne référence plus cet objet. Dans un langage comme le C++, nous aurions été obligés de vider explicitement la mémoire référencée par la variable. En C#, pas besoin ! C'est le ramasse-miettes qui le fait pour nous. Encore un truc de fainéant ça !

En fait, c'est plutôt une sécurité qu'un truc de fainéant. C'est la garantie que la mémoire utilisée par les objets qui n'existent plus est vidée et, du coup, redevient disponible pour de nouveaux objets. Grâce au ramasse-miettes, nous évitons ce que l'on appelle les fuites mémoire.



C'est super ça mais il passe à quel moment ce ramasse-miettes ?

La réponse est simple : on ne sait pas quand il passe. En fait, il passe quand il a besoin de mémoire ou quand l'application semble ne rien faire. Évidemment, lorsque le ramasse-miettes passe, les performances de l'application sont un petit peu pénalisées. C'est pour cela que l'algorithme de passage du ramasse-miettes est optimisé pour essayer de déranger le moins possible l'application, lors des moments d'inactivité par exemple. Par contre, quand il y a besoin de mémoire, il ne se pose pas la question : il passe quand même.

Il y aurait beaucoup de choses à dire encore sur ce mécanisme mais arrêtons-nous là. Retenons que le ramasse-miettes est un superbe outil qui nous permet de garantir l'intégrité de notre mémoire et qu'il s'efforce de nous embêter le moins possible. Il est important de libérer les ressources dont on n'a plus besoin quand c'est possible, par exemple en se désabonnant d'un événement ou lorsque nous avons utilisé des ressources natives (ce que nous ne verrons pas dans ce livre) ou lors d'accès à des fichiers ou des bases de données.

Masquer une méthode

Dans la partie précédente sur la substitution, nous avons vu qu'on pouvait substituer une méthode grâce au mot-clé `override`. Cette méthode devait d'ailleurs s'être déclarée comme candidate à cette substitution grâce au mot-clé `virtual`. Rappelez-vous de ce code :

```
public class Chien
{
    public virtual void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class Caniche : Chien
{
    public override void Aboyer()
    {
        Console.WriteLine("Wiiff");
    }
}
```

Si nous instancions des chiens et des caniches de cette façon :

```
static void Main(string[] args)
{
    Chien chien = new Chien();
    Caniche caniche = new Caniche();
    Chien canicheTraiteCommeUnChien = new Caniche();

    chien.Aboyer();
    caniche.Aboyer();
    canicheTraiteCommeUnChien.Aboyer();
}
```

Nous aurons :

```
Wouaf !
Wiiff
Wiiff
```

En effet, le premier objet est un chien et les deux suivants sont des caniches. Grâce à la substitution, nous faisons aboyer notre chien, notre caniche et notre caniche qui se fait manipuler comme un chien.

Il est possible de masquer des méthodes qui ne sont pas forcément marquées comme virtuelles grâce au mot-clé `new`. À ce moment-là, la méthode ne se substitue pas à la méthode dérivée mais la masque pour les types dérivés. Voyons cet exemple :

```
public class Chien
```

```
{
    public void Aboyer()
    {
        Console.WriteLine("Wouaf !");
    }
}

public class Caniche : Chien
{
    public new void Aboyer()
    {
        Console.WriteLine("Wiiff");
    }
}
```

Remarquons que la méthode `Aboyer()` de la classe `Chien` n'est pas marquée `virtual` et que la méthode `Aboyer` de la classe `Caniche` est marquée avec le mot-clé `new`. Il ne s'agit pas ici de substitution. Il s'agit juste de deux méthodes qui portent le même nom, mais la méthode `Aboyer()` de la classe `Caniche` se charge de masquer la méthode `Aboyer()` de la classe mère. Ce qui fait que les précédentes instanciations :

```
static void Main(string[] args)
{
    Chien chien = new Chien();
    Caniche caniche = new Caniche();
    Chien canicheTraiteCommeUnChien = new Caniche();

    chien.Aboyer();
    caniche.Aboyer();
    canicheTraiteCommeUnChien.Aboyer();
}
```

produiront cette fois-ci :

```
Wouaf !
Wiiff
Wouaf !
```

C'est le dernier cas qui peut surprendre. Rappelez-vous, il ne s'agit pas d'un remplacement de méthode cette fois-ci, et donc le fait de manipuler le caniche en tant que `Chien` ne permet pas de remplacer le comportement de la méthode `Aboyer()`. Dans ce cas, on appelle bien la méthode `Aboyer` correspondant au type `Chien`, ce qui a pour effet d'afficher « Wouaf ! ». Si nous n'avions pas utilisé le mot-clé `new`, nous aurions eu le même comportement mais le compilateur nous aurait avertis de ceci grâce à un avertissement :

```
'MaPremiereApplication.Caniche.Aboyer()' masque le membre hérité
    'MaPremiereApplication.Chien.Aboyer()'. Utilisez le mot-clé
   new si le masquage est intentionnel.
```

Il nous précise que nous masquons effectivement la méthode de la classe mère et que si c'est fait exprès, alors il faut l'indiquer grâce au mot-clé `new` afin de lui montrer que nous savons ce que nous faisons. Notez quand même que dans la majorité des cas, en POO, ce que vous voudrez faire, c'est bien substituer la méthode et non la masquer.

C'est le même principe pour les variables, il est également possible de masquer une variable, quoiqu'en général, cela reflète plutôt une erreur dans le code. Si nous faisons :

```
public class Chien
{
    protected int age;
}

public class Caniche : Chien
{
    private int age;

    public Caniche()
    {
        age = 0;
    }

    public override string ToString()
    {
        return "J'ai " + age + " ans";
    }
}
```

Alors le compilateur nous indique que la variable `age` de la classe `Caniche` masque la variable `age` initialement héritée de la classe `Chien`. Si c'est intentionnel, il nous propose de la marquer avec le mot clé `new`. Sinon, cela nous permet de constater que cette variable est peut-être inutile… (sûrement d'ailleurs !)

Le mot-clé `yield`

Nous avons vu dans le TP sur les génériques comment créer un énumérateur. C'est une tâche un peu lourde qui nécessite pas mal de code. Oublions un instant que la classe `String` est énumérable et créons pour l'exemple une classe qui permet d'énumérer une chaîne de caractères. Nous aurions pu faire quelque chose comme ça :

```
public class ChaineEnumerable : IEnumerable<char>
{
    private string chaine;
    public ChaineEnumerable(string valeur)
    {
        chaine = valeur;
    }

    public IEnumerator<char> GetEnumerator()
```

```
    {
        return new ChaineEnumerateur(chaine);
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new ChaineEnumerateur(chaine);
    }
}

public class ChaineEnumerateur : IEnumerator<char>
{
    private string chaine;
    private int indice;

    public ChaineEnumerateur(string valeur)
    {
        indice = -1;
        chaine = valeur;
    }

    public char Current
    {
        get { return chaine[indice]; }
    }

    public void Dispose()
    {
    }

    object IEnumerator.Current
    {
        get { return Current; }
    }

    public bool MoveNext()
    {
        indice++;
        return indice < chaine.Length;
    }

    public void Reset()
    {
        indice = -1;
    }
}
```

Vous pouvez copier ce code en utilisant le code web suivant :

▷ Copier ce code
Code web : 740399

Nous avons une classe `ChaineEnumerable` qui encapsule une chaîne de caractères de type `string` et une classe `ChaineEnumerateur` qui permet de parcourir une chaîne et de renvoyer à chaque itération un caractère, représenté par le type `char`[2].

Rien de bien sorcier, mais un code plutôt lourd.

Regardons à présent le code suivant :

```
public class ChaineEnumerable : IEnumerable<char>
{
    private string chaine;
    public ChaineEnumerable(string valeur)
    {
        chaine = valeur;
    }

    public IEnumerator<char> GetEnumerator()
    {
        for (int i = 0; i < chaine.Length; i++)
        {
            yield return chaine[i];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

Il fait la même chose, mais d'une manière bien simplifiée. Nous remarquons l'apparition du mot-clé `yield`. Il permet de créer facilement des énumérateurs. Combiné au mot-clé `return`, il renvoie un élément d'une collection et passe à l'élément suivant. Il peut être combiné au mot-clé `break` également pour permettre d'arrêter l'énumération. Ce qui veut dire que nous aurions pu écrire la méthode `GetEnumerator()` de cette façon :

```
public IEnumerator<char> GetEnumerator()
{
    int i = 0;
    while (true)
    {
        if (i == chaine.Length)
            yield break;
        yield return chaine[i];
        i++;
    }
}
```

Ce qui est plus laid, je le conçois ! mais qui permet d'illustrer le mot-clé `yield break`.

2. Le type `char` est un alias de la structure `System.Char`.

Le mot-clé `yield` permet également de renvoyer un `IEnumerable` (ou sa version générique), ainsi un peu bêtement, on pourrait faire :

```
static void Main(string[] args)
{
    foreach (string prenom in ObtenirListeDePrenoms())
    {
        Console.WriteLine(prenom);
    }
}

public static IEnumerable<string> ObtenirListeDePrenoms()
{
    yield return "Nicolas";
    yield return "Jérémie";
    yield return "Delphine";
}
```

Cet exemple va surtout me permettre d'illustrer ce qui se passe exactement grâce au mode debug. Mettez un point d'arrêt au début du programme et lancez-le en mode debug (voir la figure 34.5).

FIGURE 34.5 – Le programme est en pause sur le point d'arrêt avant l'énumération

Appuyez sur F11 plusieurs fois pour rentrer dans la méthode `ObtenirListeDePrenoms`, nous voyons l'indicateur se déplacer sur les différentes parties de l'instruction `foreach`. Puis nous arrivons sur la première instruction `yield return` (voir la figure 34.6).

Appuyons à nouveau sur F11 et nous pouvons constater que nous sortons de la méthode et que nous rentrons à nouveau dans le corps de la boucle `foreach` qui va nous afficher le premier prénom. Continuons les F11 jusqu'à repasser sur le mot-clé `in` et rentrer à nouveau dans la méthode `ObtenirListeDePrenoms()` et nous pouvons constater que nous retournons directement au deuxième mot-clé `yield`, ainsi que vous pouvez le voir à la figure 34.7.

Et ainsi jusqu'à ce qu'il n'y ait plus rien dans la méthode `ObtenirListeDePrenoms()` et que, du coup, le `foreach` se termine. À la première lecture, ce n'est pas vraiment ce comportement que nous aurions attendu...

```
namespace MaPremiereApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            foreach (string prenom in ObtenirListeDePrenoms())
            {
                Console.WriteLine(prenom);
            }
        }

        public static IEnumerable<string> ObtenirListeDePrenoms()
        {
            yield return "Nicolas";
            yield return "Jérémie";
            yield return "Delphine";
        }
    }
```

100 %

FIGURE 34.6 – Le programme atteint l'instruction `yield` juste après le `foreach`

```
namespace MaPremiereApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            foreach (string prenom in ObtenirListeDePrenoms())
            {
                Console.WriteLine(prenom);
            }
        }

        public static IEnumerable<string> ObtenirListeDePrenoms()
        {
            yield return "Nicolas";
            yield return "Jérémie";
            yield return "Delphine";
        }
    }
```

FIGURE 34.7 – Le programme atteint la seconde instruction `yield` lors de la deuxième itération

Voilà pour le principe du `yield return`.



Notez qu'un `yield break` nous aurait fait sortir directement de la boucle `foreach`.

Le mot-clé `yield` participe à ce qu'on appelle **l'exécution différée**. On n'évalue la méthode `ObtenirListeDePrenoms()` qu'au moment où on parcoure le résultat avec la boucle `foreach`, ce qui pourrait paraître surprenant. Par exemple, si nous faisons :

```
public static IEnumerable<string> ObtenirListeDePrenoms()
{
    yield return "Nicolas";
    yield return "Jérémie";
    yield return "Delphine";
}

static void Main(string[] args)
{
    IEnumerable<string> prenoms = ObtenirListeDePrenoms();
    Console.WriteLine("On fait des choses ...");
    foreach (string prenom in prenoms)
    {
        Console.WriteLine(prenom);
    }
}
```

et que nous mettons un point d'arrêt sur le `Console.WriteLine` et un autre dans la méthode `ObtenirListeDePrenoms()`, on s'arrêtera d'abord sur le point d'arrêt de la ligne où est situé le `Console.WriteLine` et nous passerons ensuite dans le point d'arrêt positionné dans la méthode `ObtenirListeDePrenoms()`. En effet, on ne passera dans la méthode que lorsqu'on itérera explicitement sur les prénoms grâce au `foreach`. Nous reviendrons sur cette exécution différée un peu plus loin avec le mot-clé `yield`.

Le formatage de chaînes, de dates et la culture

Pour l'instant, lorsque nous avons eu besoin de mettre en forme des chaînes de caractères, nous avons utilisé la méthode `Console.WriteLine` avec en paramètres des chaînes concaténées entre elles grâce à l'opérateur +. Par exemple :

```
int age = 30;
Console.WriteLine("J'ai " + age + " ans");
```

Il est important de savoir qu'il est possible de formater la chaîne un peu plus simplement et surtout beaucoup plus efficacement. Ceci est possible grâce à la méthode `string.Format`. Le code précédent peut par exemple être remplacé par :

```
int age = 30;
```

```
string chaine = string.Format("J'ai {0} ans", age);
Console.WriteLine(chaine);
```

La méthode `string.Format` accepte en premier paramètre un format de chaîne. Ici, nous lui indiquons une chaîne de caractères avec au milieu un caractère spécial : `{0}`. Il permet de dire : « remplace-moi le `{0}` avec le premier paramètre qui va suivre », en l'occurrence la variable `age`. Si nous avons plusieurs valeurs, il est possible d'utiliser les caractères spéciaux `{1}`, puis `{2}`, etc. Chaque nombre représente ici l'indice correspondant au paramètre. Par exemple :

```
double valeur1 = 10.5;
double valeur2 = 4.2;
string chaine = string.Format("{0} multiplié par {1} s'écrit
    \"{0} * {1}\" et donne comme résultat : {2}", valeur1,
    valeur2, valeur1 * valeur2);
Console.WriteLine(chaine);
```

Ce qui donne :

```
10,5 multiplié par 4,2 s'écrit "10,5 * 4,2" et donne comme ré
   sultat : 44,1
```

Vous avez pu remarquer qu'il est possible d'utiliser le même paramètre à plusieurs endroits. La méthode `Console.WriteLine` dispose aussi de la capacité de formater des chaînes de caractères. Ce qui veut dire que le code précédent peut s'écrire :

```
double valeur1 = 10.5;
double valeur2 = 4.2;
Console.WriteLine("{0} multiplié par {1} s'écrit \"{0} * {1}\"
    et donne comme résultat : {2}", valeur1, valeur2, valeur1 *
    valeur2);
```

Ce qui revient absolument au même.

Parlons culture désormais. N'ayez pas peur, rien à voir avec des questions pour des champions ! La culture en informatique correspond à tout ce qui a trait aux différences entre les langues et les paramètres locaux. Par exemple, en France et aux États-Unis, les dates s'écrivent différemment. Pour le jour de Noël 2011, voici ce que ça donne :

- en France : le 25/12/2011 ;
- aux États-Unis : the 12/25/2011.

Le mois et le jour sont donc inversés. De la même façon, il existe des différences lorsque nous écrivons les nombres à virgule :

- en France : 123,45 ;
- aux États-Unis : 123.45.

Un point à la place d'une virgule, subtile différence. On a donc souvent des différences entre les langues. Et même plus, on peut avoir des différences entre les langues en fonction de l'endroit où elles sont parlées. Citons par exemple le français de France et le français du Canada. Il existe donc au sein du système d'exploitation tout une liste de

« régions » représentées par un couple de lettres. Par exemple, le français est représenté par les lettres « fr ». Et même plus précisément, le français de France est représenté par « fr-FR » alors que celui du Canada est représenté par « fr-CA ». C'est ce que l'on retrouve dans les paramètres de notre système d'exploitation lorsqu'on va (sous Windows 7) dans le panneau de configuration, `Horloge langue et région`, comme vous pouvez le voir à la figure 34.8.

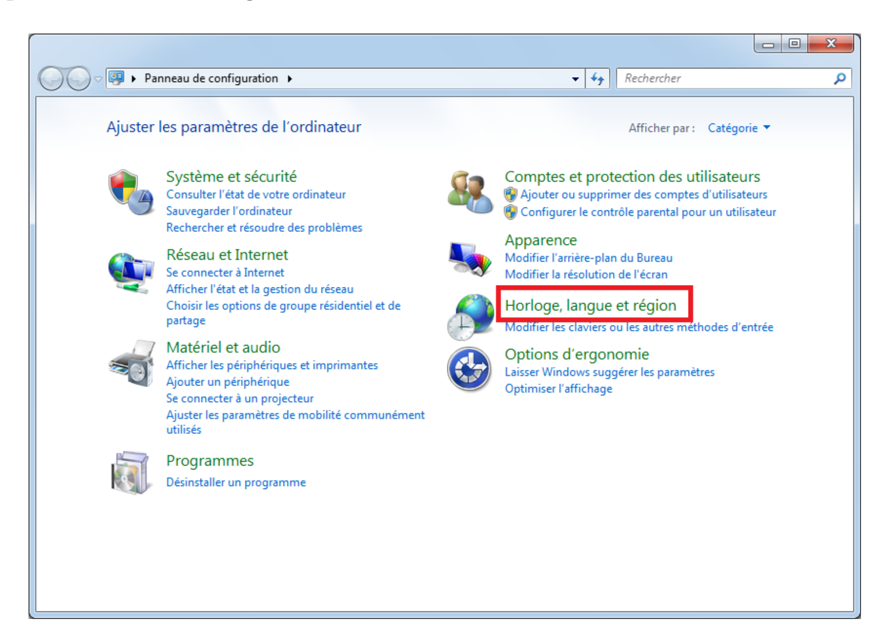


FIGURE 34.8 – Choisir Horloge, langue et région dans le panneau de configuration

Si vous allez dans `Modifier le format de la date, de l'heure ou des nombres` (voir figure 34.9), vous pouvez adapter le format à celui que vous préférez (voir la figure 34.10).

Bref, ce choix, nous le retrouvons dans notre application si nous récupérons la culture courante :

```
Console.WriteLine(System.Threading.Thread.CurrentThread.
    CurrentCulture);
```

Avec ceci, nous aurons :

```
fr-FR
```

Ce qui est intéressant, c'est que le formatage des chiffres ou des dates change en fonction de la culture. Ainsi, si nous utilisons le français de France, pour le code suivant :

```
public static void Main(string[] args)
{
    Affiche();
```

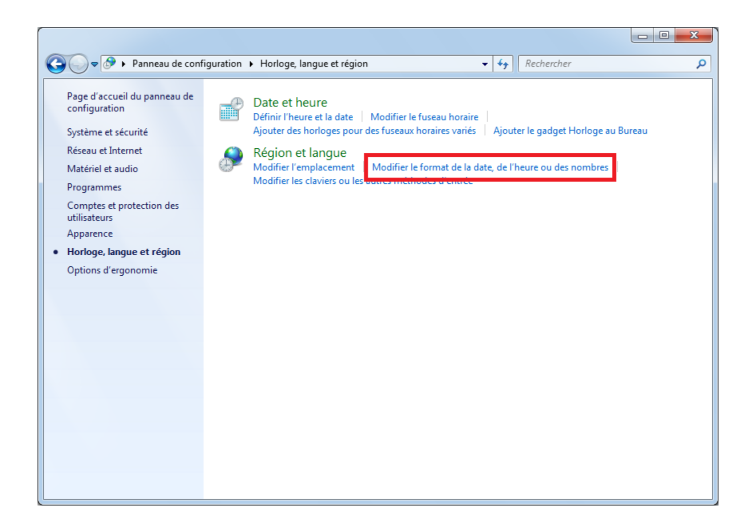


FIGURE 34.9 – On peut modifier ici le format de la date

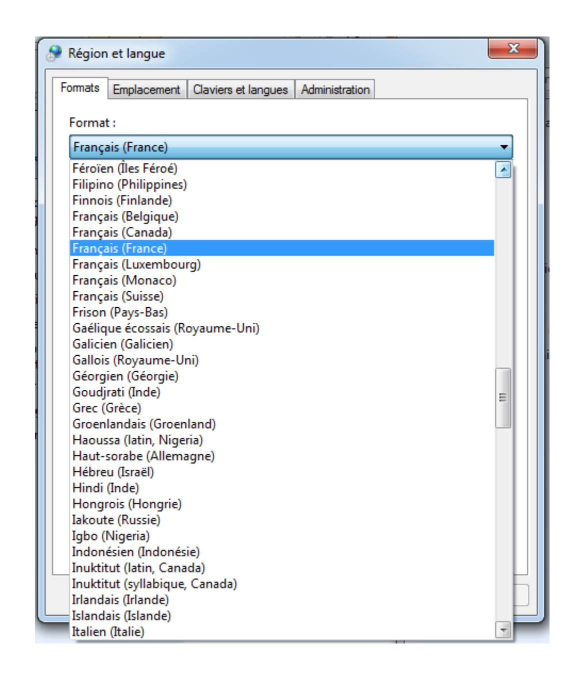


FIGURE 34.10 – Sélection de la langue pour le format

```
}

public static void Affiche()
{
    Console.WriteLine(System.Threading.Thread.CurrentThread.
        CurrentCulture);
    decimal dec = 5.5M;
    double dou = 4.8;
    DateTime date = new DateTime(2011, 12, 25);
    Console.WriteLine("Décimal : {0}", dec);
    Console.WriteLine("Double : {0}", dou);
    Console.WriteLine("Date : {0}", date);
}
```

nous aurons :

```
fr-FR
Décimal : 5,5
Double : 4,8
Date : 25/12/2011 00:00:00
```

Il est possible de changer la culture courante de notre application, si par exemple nous souhaitons qu'elle soit multiculture, en utilisant le code suivant :

```
System.Threading.Thread.CurrentThread.CurrentCulture = new
    CultureInfo("en-US");
```

Ici par exemple, j'indique à mon application que je souhaite travailler avec la culture des États-Unis. Ainsi, le code précédent, en changeant juste la culture :

```
public static void Main(string[] args)
{
    System.Threading.Thread.CurrentThread.CurrentCulture = new
        CultureInfo("en-US");
    Affiche();
}

public static void Affiche()
{
    Console.WriteLine(System.Threading.Thread.CurrentThread.
        CurrentCulture);
    decimal dec = 5.5M;
    double dou = 4.8;
    DateTime date = new DateTime(2011, 12, 25);
    Console.WriteLine("Décimal : {0}", dec);
    Console.WriteLine("Double : {0}", dou);
    Console.WriteLine("Date : {0}", date);
}
```

produira un résultat légèrement différent :

```
en-US
Décimal : 5.5
Double : 4.8
Date : 12/25/2011 12:00:00 AM
```

Il s'agit exactement de la même information, mais formatée différemment.

À noter qu'il existe encore une autre information de culture, à savoir la langue du système d'exploitation que l'on retrouve dans la culture de l'interface graphique, dans la propriété :

```
Console.WriteLine(System.Threading.Thread.CurrentThread.
    CurrentUICulture);
```

Il s'agit de la `CurrentUICulture`, à ne pas confondre avec la `CurrentCulture`. Correspondant à la langue du système d'exploitation, nous nous en servirons plus volontiers pour rendre une application multilingue.

Pour finir sur le formatage des chaînes, il est possible d'utiliser des caractères spéciaux enrichis pour changer le formatage des types numériques ou de la date. Par exemple, il est possible d'afficher la date sous une forme différente avec :

```
DateTime date = new DateTime(2011, 12, 25);
Console.WriteLine("La date est {0}", date.ToString("dd-MM-yyyy"
    ));
```

Ici, « dd » sert à afficher le jour, « MM » le mois et « yyyy » l'année sur quatre chiffres, ce qui donne :

```
La date est 25-12-2011
```

Il y a beaucoup d'options différentes que vous pouvez retrouver dans la documentation Microsoft, via le code web suivant :

▷ `DateTime`
Code web : 781180

Le fonctionnement est similaire pour les numériques ; ce qui nous permet par exemple d'afficher un nombre de différentes manières.

Ici par exemple, j'utilise la notation scientifique :

```
double valeur = 123.45;
Console.WriteLine("Format scientifique : {0:e}", valeur);
```

Ce qui donne :

```
Format scientifique : 1,234500e+002
```

Pour les nombres, il existe différents formatages que l'on peut retrouver dans le tableau ci-après.

Format	Description	Remarque
{0 :c}	Monnaie	Attention, la console affiche mal le caractère €
{0 :d}	Décimal	Ne fonctionne qu'avec les types sans virgules
{0 :e}	Notation scientifique	
{0 :f}	Virgule flottante	
{0 :g}	Général	Valeur par défaut
{0 :n}	Nombre avec formatage pour les milliers	
{0 :r}	Aller-retour	Permet de garantir que la valeur numérique convertie en chaîne pourra être convertie à nouveau en valeur numérique identique
{0 :x}	Hexadécimal	Ne fonctionne qu'avec les types sans virgules

On peut également avoir du formatage grâce à la méthode `ToString()`. Il suffit de passer le format en paramètres, comme ici, où cela me permet d'afficher uniquement le nombre de chiffres après la virgule qui m'intéressent ou un nombre des chiffres significatifs :

```
double valeur = 10.0 / 3;
Console.WriteLine("Avec 3 chiffres significatifs et 3 chiffres
    après la virgule : {0}", valeur.ToString("000.###"));
```

Qui donne :

```
Avec 3 chiffres significatifs et 3 chiffres après la virgule :
   003,333
```

Mettre le format dans la méthode `ToString()` fonctionne aussi pour la date :

```
DateTime date = new DateTime(2011, 12, 25);
Console.WriteLine(date.ToString("MMMM"));
```

Ici, je n'affiche que le nom du mois :

```
décembre
```

Voilà pour le formatage des chaînes de caractères. Lister tous les formatages possibles serait une tâche bien ennuyeuse. Vous aurez l'occasion de rencontrer d'autres formatages qui permettent de faire un peu tout et n'importe quoi. Vous pourrez retrouver la plupart des formatages à partir de ce code web :

▷ Formatages
Code web : 500571

Les attributs

Dans la liste des choses qui vont vous servir, on peut ajouter les attributs. Ils sont utilisés pour fournir des informations complémentaires sur une méthode, une classe, des variables, etc. Ils vont nous servir régulièrement dans le framework .NET et nous aurons même la possibilité de créer nos propres attributs. Un attribut est déclaré entre crochets avec le nom de sa classe. Il peut se mettre au-dessus d'une classe, d'une méthode, d'une propriété, etc. Par exemple, dans le code ci-dessous, je positionne l'attribut `Obsolete` au-dessus de la déclaration d'une méthode :

```
[Obsolete("Utilisez plutôt la méthode ToString() pour avoir une
    représentation de l'objet")]
public void Affiche()
{
    // ...
}
```

Ici, j'utilise un attribut existant du framework .NET, l'attribut `Obsolete` qui permet d'indiquer qu'une méthode est obsolète et qu'il vaudrait mieux ne pas l'utiliser. En général, il est possible de fournir une information permettant de dire pourquoi la méthode est obsolète et par quoi il faut la remplacer. Ainsi, si nous tentons d'utiliser cette méthode :

```
public class Program
{
    static void Main(string[] args)
    {
        new Program().Affiche();
    }

    [Obsolete("Utilisez plutôt la méthode ToString() pour avoir
        une représentation de l'objet")]
    public void Affiche()
    {
        // ...
    }
}
```

Le compilateur nous affichera l'avertissement suivant :

```
'MaPremiereApplication.Program.Affiche()' est obsolète : '
   Utilisez plutôt la méthode ToString() pour avoir une repré
   sentation de l'objet'
```

Cet attribut est un exemple, il en existe beaucoup dans le framework .NET et vous aurez l'occasion d'en voir d'autres dans les chapitres suivants. Remarquons que la classe s'appelle `ObsoleteAttribute` et peut s'utiliser soit suffixée par `Attribute`, soit sans, et dans ce cas, ce suffixe est ajouté automatiquement.

Grâce à l'héritage, nous allons nous aussi pouvoir définir nos propres attributs. Il suffit pour cela de dériver de la classe de base `Attribute`. Par exemple, dans sa forme la plus

simple :

```
public class DescriptionClasseAttribute : Attribute
{
}
```

Nous pourrons alors utiliser notre attribut sur une classe :

```
[DescriptionClasse]
public class Chien
{
}
```

Super, même si ça ne nous sert encore à rien !

Problème, j'ai appelé mon attribut `DescriptionClasse` et je peux quand même l'utiliser sur une méthode :

```
[DescriptionClasse]
public class Chien
{
    [DescriptionClasse]
    public void Aboyer()
    {
    }
}
```

Heureusement, il est possible d'indiquer des restrictions sur les attributs en indiquant par exemple qu'ils ne sont valables que pour les classes. Cela se fait avec... un attribut !

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class DescriptionClasseAttribute : Attribute
{
}
```

Ici par exemple, j'indique que mon attribut n'est valable que sur les classes et qu'il est possible de le mettre plusieurs fois sur la classe. Le code précédent où l'attribut est positionné sur une méthode ne compilera donc plus, avec l'erreur de compilation suivante :

```
L'attribut 'DescriptionClasse' n'est pas valide dans ce type de
    déclaration. Il n'est valide que dans les déclarations 'class
    '.
```

Ce qui nous convient parfaitement !



Il existe d'autres attributs qui permettent par exemple de n'autoriser des attributs que sur les méthodes.

Il est intéressant de pouvoir fournir des informations complémentaires à notre attribut. Pour cela, on peut rajouter des propriétés, ou avoir une surcharge complémentaire du constructeur, comme on l'a fait pour le message de l'attribut `Obsolete` :

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class DescriptionClasseAttribute : Attribute
{
    public string Description { get; set; }

    public DescriptionClasseAttribute()
    {
    }

    public DescriptionClasseAttribute(string description)
    {
        Description = description;
    }
}
```

Ce qui nous permettra d'utiliser notre attribut de ces deux façons :

```
[DescriptionClasse(Description = "Cette classe correspond à un
    chien")]
[DescriptionClasse("Elle dérive de la classe Animal")]
public class Chien : Animal
{
}
```

C'est très bien ces attributs, mais nous ne sommes pas encore capables de les exploiter. Voyons maintenant comment le faire.

La réflexion

La réflexion en C# ne donne pas mal à la tête, quoique. . . ! C'est une façon de faire de l'introspection sur nos types de manière à obtenir des informations sur eux, c'est-à-dire sur les méthodes, les propriétés, etc. La réflexion permet également de charger dynamiquement des types et de faire de l'introspection sur les assemblys. C'est un mécanisme assez complexe et plutôt coûteux en terme de temps. Le point de départ est la classe `Type`, qui permet de décrire n'importe quel type. Le type d'une classe peut être obtenu grâce au mot-clé `typeof` :

```
Type type = typeof(string);
```

Cet objet `Type` nous permet d'obtenir plein d'informations sur nos classes au moment de l'exécution. Par exemple si le type est une classe, c'est-à-dire ni un type valeur, ni une interface, alors le code suivant :

```
Type type = typeof(string);
Console.WriteLine(type.IsClass);
```

nous renverra `true`. En effet, la propriété `IsClass` permet d'indiquer si le type est une classe. Nous pouvons aussi obtenir le nom de méthodes grâce à `GetMethods()` :

```
Type type = typeof(string);
foreach (MethodInfo infos in type.GetMethods())
{
    Console.WriteLine(infos.Name);
}
```

Je vous renvoie à la figure 34.11 pour le résultat dans la console.

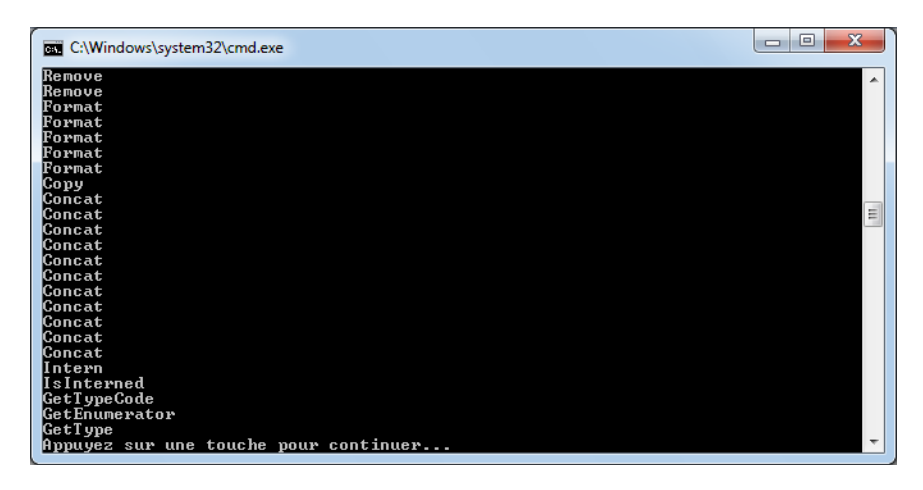


Figure 34.11 – Affichage du nom des méthodes

Pas très exploitable comme résultat, mais bon, c'est pour l'exemple ! Il y a encore pleins d'infos dans cet objet `Type`, qui permettent d'obtenir ce que nous voulons. Cela est possible grâce aux métadonnées qui sont des informations de description des types.

Vous me voyez venir. . . eh oui, nous allons pouvoir obtenir nos attributs. Pour connaître la liste des attributs d'un type, on peut utiliser la méthode statique `Attribute.GetCustomAttributes()`, en lui passant en paramètre le `System.Type` qui est hérité de la classe abstraite `MemberInfo`. Si l'on souhaite récupérer un attribut particulier, on peut le passer en paramètre de la méthode :

```
Attribute[] lesAttributs = Attribute.GetCustomAttributes(type,
    typeof(DescriptionClasseAttribute));
```

Ceci va nous permettre de récupérer les attributs de notre classe et en l'occurrence, d'obtenir leurs descriptions. C'est ce que permet le code suivant :

```
public class Program
{
    static void Main(string[] args)
    {
        new DemoAttributs().Demo();
    }

}

```

```
public class DemoAttributs
{
    public void Demo()
    {
        Animal animal = new Animal();
        Chien chien = new Chien();

        VoirDescription(animal);
        VoirDescription(chien);
    }

    public void VoirDescription<T>(T obj)
    {
        Type type = typeof(T);
        if (!type.IsClass)
            return;
        Attribute[] lesAttributs = Attribute.GetCustomAttributes(type, typeof(DescriptionClasseAttribute));
        if (lesAttributs.Length == 0)
            Console.WriteLine("Pas de description pour la classe " + type.Name + "\n");
        else
        {
            Console.WriteLine("Description pour la classe " + type.Name);
            foreach (DescriptionClasseAttribute attribut in lesAttributs)
            {
                Console.WriteLine("\t" + attribut.Description);
            }
        }
    }
}
```

Nous commençons par instancier un objet `animal` et un objet `chien`. Puis nous demandons leurs descriptions. La méthode générique `VoirDescription()` s'occupe dans un premier temps d'obtenir le type de la classe grâce à `typeof`. On s'octroie une vérification permettant de nous assurer que le type est bien une classe. Ceci permet d'éviter d'aller plus loin car de toute façon, notre attribut ne s'applique qu'aux classes. Puis nous utilisons la méthode `GetCustomAttributes` pour obtenir les attributs de type `DescriptionClasseAttribute`. S'il y en a, alors on les affiche.

Ce qui donne :

```
Pas de description pour la classe Animal

Description pour la classe Chien
Elle dérive de la classe Animal
Cette classe correspond à un chien
```

Voilà pour les attributs, vous aurez l'occasion de rencontrer des attributs du framework .NET un peu plus loin dans ce tutoriel, mais globalement, il est assez rare d'avoir à créer soi-même ses attributs.

En ce qui concerne la réflexion, il faut bien comprendre que le sujet est beaucoup plus vaste que ça. Nous avons cependant vu ici un aperçu de ce que permet de faire la réflexion, en illustrant le mécanisme d'introspection sur les attributs.

En résumé

- Il est possible d'empêcher une classe d'être dérivée grâce au mot-clé `sealed`.
- Le garbage collector est le mécanisme permettant de libérer la mémoire allouée sur le tas managé qui n'est plus référencée dans une application.
- Le mot-clé `yield` permet de créer des énumérateurs facilement et participe au mécanisme d'exécution différée.
- Le framework .NET gère les différents formats des chaînes grâce à la culture de l'application.
- La réflexion est un mécanisme d'introspection sur les types permettant d'obtenir des informations sur ces derniers lors de l'exécution.

Chapitre 35

La configuration d'une application

Difficulté : 

Dans le cycle de vie d'une application, il est fréquent que de petites choses changent. Imaginons que mon application doive se connecter à un serveur FTP pour sauvegarder les données sur lesquelles j'ai travaillé. Pendant mes tests, effecutés en local, mon serveur FTP pourrait avoir l'adresse `ftp://localhost` avec un login et un mot de passe « test ». Évidemment, lors de l'utilisation réelle de mon application, l'adresse changera, et le login et le mot de passe varieront en fonction de l'utilisateur. Il faut être capable de changer facilement ces paramètres sans avoir à modifier le code ni à recompiler l'application.

C'est là qu'interviennent les fichiers de configuration : ils permettent de stocker toutes ces petites choses qui servent à faire varier notre application. Pour les clients lourds, comme nos applications console, il s'agit du fichier `app.config` : un fichier XML qui contient la configuration de notre application. Il stocke des chaînes de connexion à une base de données, une url de service web, des préférences utilisateurs, etc. Bref, tout ce qui va permettre de configurer notre application !



Rappel sur les fichiers XML

Vous ne connaissez pas les fichiers XML ? Si vous voulez en savoir plus, n'hésitez pas à faire un petit tour sur internet, c'est un format très utilisé dans l'informatique. Pour faire court, le XML est un langage de balise, un peu comme le HTML, où l'on décrit de l'information. Les balises sont des valeurs entourées de < et > qui décrivent la sémantique de la donnée. Par exemple :

```
<prenom>Nicolas</prenom>
```

La balise `<prenom>` est ce qu'on appelle une balise ouvrante, cela signifie que ce qui se trouve après (en l'occurrence la chaîne « Nicolas ») fait partie de cette balise jusqu'à ce que l'on rencontre la balise fermante `</prenom>` qui est comme la balise ouvrante à l'exception du / précédant le nom de la balise. Le XML est un fichier facile à lire par nous autres humains. On en déduit assez facilement que le fichier contient la chaîne « Nicolas » et qu'il s'agit sémantiquement d'un prénom. Une balise peut contenir des attributs permettant de donner des informations sur la donnée. Les attributs sont entourés de guillemets : `""` et font partie de la balise. Par exemple :

```
<client nom="Nicolas" age="30"></client>
```

Ici, la balise client possède un attribut `nom` ayant la valeur « Nicolas » et un attribut `age` ayant la valeur « 30 ». Encore une fois, c'est très facile à lire pour un humain. Il est possible que la balise n'ait pas de valeur, comme c'est le cas dans l'exemple ci-dessus. On peut dans ce cas-là remplacer la balise ouvrante et la balise fermante par cet équivalent :

```
<client nom="Nicolas" age="30"/>
```

Enfin, et nous allons terminer notre aperçu rapide du XML avec ce dernier point, il est important de noter que le XML peut imbriquer ses balises et qu'il peut ne posséder qu'un seul élément racine, ce qui nous permet d'avoir une hiérarchie de données. Par exemple nous pourrons avoir :

```
<listesDesClient>
  <client type="Particulier">
    <nom>Nicolas</nom>
    <age>30</age>
  </client>
  <client type="Professionel">
    <nom>Jérémie</nom>
    <age>40</age>
  </client>
</listesDesClient>
```

On voit tout de suite que le fichier décrit une liste de deux clients. Nous en avons un qui est un particulier, qui s'appelle Nicolas et qui a 30 ans alors que l'autre est un professionnel, prénommé Jérémie et qui a 40 ans.

Créer le fichier de configuration

Pourquoi utiliser un fichier de configuration ?

– Pour éviter de mettre des valeurs en dur dans le code. Imaginons que nous utilisions une url de service web dans notre application, si l'url change, on aimerait ne pas avoir à recompiler le code.
– Pour éviter d'utiliser la base de registre comme cela a beaucoup été fait dans les premières versions de Windows et qui oblige à donner les droits de modification sur celle-ci.

Ces fichiers permettent d'avoir des informations de configuration, potentiellement typées. De plus, le framework .NET dispose de méthodes pour y accéder facilement. L'intérêt d'utiliser un fichier XML plutôt qu'un fichier binaire est que ce fichier est lisible et compréhensible facilement. On peut également le modifier à la main sans un système évolué permettant de faire des modifications. Voilà plein de raisons pour lesquelles on va utiliser ces fichiers.

Pour ajouter un fichier de configuration : faisons un clic droit sur le projet : `Ajouter` > `Nouvel élément`, comme l'illustre la figure 35.1.

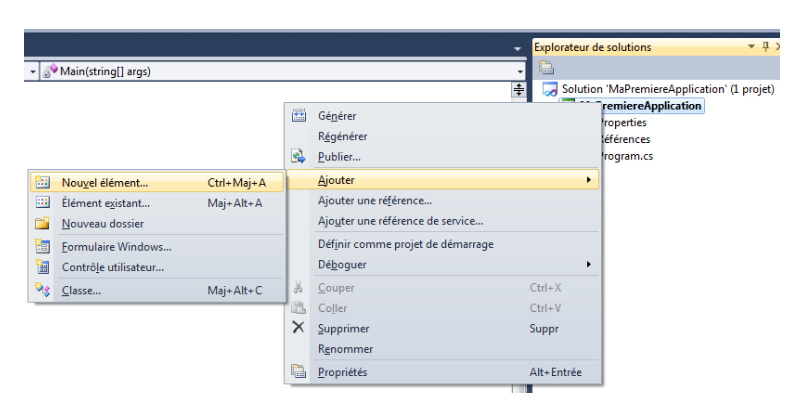


FIGURE 35.1 – Ajout du fichier de configuration

Choisissons le modèle de fichier `Fichier de configuration de l'application` (voir figure 35.2).

Conservez son nom et validez sa création. Ce fichier est presque vide :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

Il possède sur sa première ligne un marqueur permettant d'identifier le fichier comme étant un fichier XML et une balise ouvrante **configuration**, suivie de sa balise fermante. C'est la structure de base du fichier de configuration. Nous mettrons nos sections de configuration à l'intérieur de cette balise. Nous retrouvons notre fichier de configuration dans notre projet et nous pouvons le voir dans l'explorateur de documents, ainsi que l'illustre la figure 35.3.

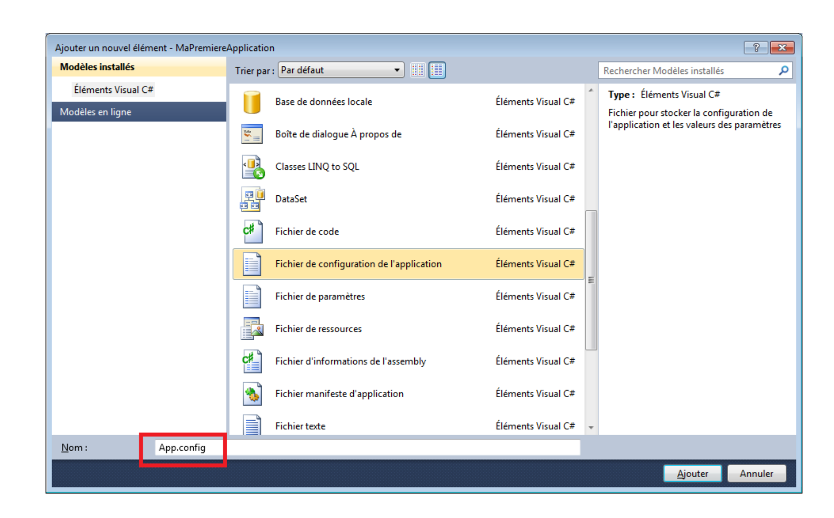


FIGURE 35.2 – Choix et nommage du fichier de configuration

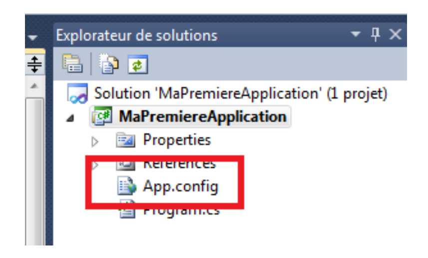


FIGURE 35.3 – Le fichier de configuration est présent dans le projet

Compilons notre application et rendons-nous dans le répertoire où le fichier exécutable est généré. Nous y trouvons également un fichier qui porte le même nom que notre exécutable et dont l'extension est `.exe.config`. C'est bien notre fichier de configuration. Visual C# Express le déploie au même endroit que notre application et lui donne un nom qui va lui permettre d'être exploité par notre application. Pour être utilisables, ces fichiers doivent se situer dans le même répertoire que l'exécutable.

Lecture simple dans la section de configuration prédéfinie : `AppSettings`

Avoir un fichier de configuration, c'est bien. Mais il faut savoir lire à l'intérieur si nous souhaitons l'exploiter. Il existe plusieurs façons d'indiquer des valeurs de configuration, nous les découvrirons tout au long de ce chapitre. La plus simple est d'utiliser la section `AppSettings`. C'est une section où l'on peut mettre, comme son nom le suggère aux anglophones, les propriétés de l'application. Pour cela, on utilisera un système de clé ou valeur pour stocker les informations. Par exemple, modifiez le fichier de configuration pour avoir :

```
<configuration>
  <appSettings>
    <add key="prenom" value="nicolas"/>
    <add key="age" value="30"/>
  </appSettings>
</configuration>
```

En voyant ce fichier de configuration, un être humain comprendra assez facilement qu'il y a deux paramètres de configuration. Un premier qui est le prénom et qui va valoir Nicolas. Un deuxième qui est l'âge et qui sera de 30. Le savoir en tant qu'être humain, c'est bien. Pouvoir y accéder dans notre programme informatique, c'est mieux ! Voyons comment faire. La première chose à faire est de référencer, si ce n'est déjà fait, l'assembly qui contient toutes les classes permettant de gérer la configuration. Elle s'appelle (sans surprise) `System.Configuration` (voir figure 35.4).

Afin d'accéder au fichier de configuration, nous allons devoir utiliser la classe statique `ConfigurationManager`. Pour accéder aux informations contenues dans la section `AppSettings`, nous utiliserons sa propriété `AppSettings`. Et nous pourrons accéder aux éléments de la configuration en utilisant l'opérateur d'indexation : []. Ce qui donne :

```
string prenom = ConfigurationManager.AppSettings["prenom"];
string age = ConfigurationManager.AppSettings["age"];

Console.WriteLine("Prénom : " + prenom + " / Age : " + age);
```

On peut également utiliser des index numériques pour y accéder, mais je trouve que ça manque vraiment de clarté :

```
string prenom = ConfigurationManager.AppSettings[0];
string age = ConfigurationManager.AppSettings[1];
```

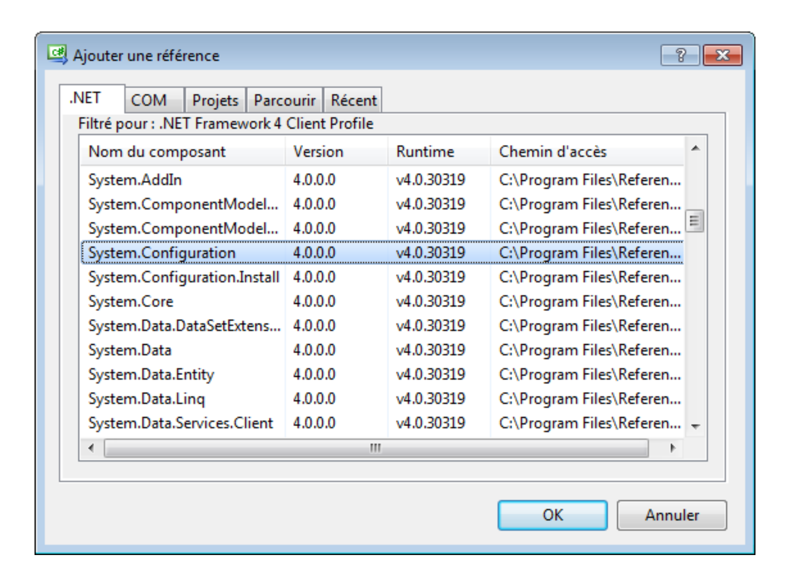


FIGURE 35.4 – Ajout de la référence à l'assembly `System.Configuration`

```

Console.WriteLine("Prénom : " + prenom + " / Age : " + age);
```

De plus, cela oblige à connaître l'ordre dans lequel les clés ont été mises dans le fichier. Il est possible aussi de boucler sur toutes les valeurs contenues dans la section :

```
foreach (string cle in ConfigurationManager.AppSettings)
{
    Console.WriteLine("Clé : " + cle + " / Valeur : " +
        ConfigurationManager.AppSettings[cle]);
}
```

La propriété `AppSettings` est en fait du type `NameValueCollection`. Il s'agit d'une collection d'éléments de type chaîne de caractères qui sont accessibles à partir d'une clé de type chaîne de caractères également. Une valeur est donc associée à une clé.

Pour plus d'informations sur le type `NameValueCollection`, je vous invite à consulter la documentation en ligne, via le code web suivant :

▷ `NameValueCollection`
Code web : 357840

À noter que la casse de la clé n'est pas importante pour accéder à la valeur associée à la clé. Le code suivant :

```
string prenom = ConfigurationManager.AppSettings["PRENOM"];
```

renverra bien notre valeur de configuration. Cependant, si la valeur n'existe pas, nous obtiendrons la valeur `null` dans la chaîne de caractères. Il est important de remarquer que nous récupérons des chaînes et que nous aurons besoin potentiellement de faire une

conversion pour manipuler l'âge en tant qu'entier. Rien ne vous empêche d'écrire une petite méthode d'extension maintenant que vous savez faire :

```
public static class ConfigurationManagerExtensions
{
    public static int ObtenirValeurEntiere(this
        NameValueCollection appSettings, string cle)
    {
        string valeur = appSettings[cle];
        return Convert.ToInt32(valeur);
    }
}
```

Que nous pourrons utiliser avec :

```
int age = ConfigurationManager.AppSettings.ObtenirValeurEntiere
    ("age");
```

Lecture des chaînes de connexion à la base de données

Les chaînes de connexion représentent un type de configuration particulier. Elles vont servir pour les applications ayant besoin de se connecter à une base de données. On va y stocker tout ce dont on a besoin, comme le nom du serveur ou les identifiants pour s'y connecter... Nous y reviendrons en détail plus tard, mais regardons la configuration suivante :

```
<configuration>
  <connectionStrings>
    <add name="MaConnection" providerName="System.Data.
        SqlClient"
      connectionString="Data Source=.\SQLEXPRESS; Initial
          Catalog=Base1; Integrated Security=true"/>
    <add name="MaConnection2" providerName="System.Data.
        SqlClient"
      connectionString="Data Source=.\SQLEXPRESS; Initial
          Catalog=Base2; Integrated Security=true"/>
  </connectionStrings>
</configuration>
```

Nous définissons ici deux chaînes de connexion qui permettent de se connecter en authentification Windows (`Integrated Security=true`) sur un serveur hébergé en local (`.\SQLEXPRESS`) dont le nom est SQLEXPRESS, qui utilisent SQL SERVER (`providerName="System.Data.SqlClient"`), qui pointent sur les bases de données `Base1` et `Base2`, identifiées chacune par les noms `MaConnection` et `MaConnection2`. Pour obtenir les informations de configuration individuellement, il faudra utiliser la propriété `ConnectionStrings` de la classe `ConfigurationManager` en y accédant par leurs noms :

```
ConnectionStringSettings chaineConnexion = ConfigurationManager
    .ConnectionStrings["MaConnection"];
Console.WriteLine(chaineConnexion.Name);
Console.WriteLine(chaineConnexion.ConnectionString);
Console.WriteLine(chaineConnexion.ProviderName);
```

Nous pourrons toutes les obtenir avec une boucle `foreach` :

```
foreach (ConnectionStringSettings valeur in
    ConfigurationManager.ConnectionStrings)
{
    Console.WriteLine(valeur.ConnectionString);
}
```

Nous utiliserons les chaînes de connexion dans le chapitre sur l'accès aux données.

Créer sa section de configuration depuis un type prédéfini

Il est possible de créer sa propre section de configuration à partir d'un type prédéfini. Par exemple pour créer une section du même genre que la section `appSettings`, qui utilise une paire clé / valeur, on peut utiliser un `DictionarySectionHandler`. Il existe plusieurs types prédéfinis, que nous allons étudier ci-dessous.

Le type `DictionarySectionhandler`

`DictionarySectionhandler` est une classe qui fournit les informations de configuration des paires clé / valeur d'une section de configuration.



Oui mais si tu nous dis que c'est un système de clé / valeur, c'est comme pour les `AppSettings` que nous avons vus. Pourquoi utiliser une section particulière `DictionarySectionhandler` alors ?

L'intérêt de pouvoir faire des sections particulières est d'organiser sémantiquement son fichier de configuration, pour découper logiquement son fichier au lieu d'avoir tout dans la même section. Regardons cette configuration :

```
<configuration>
  <configSections>
    <section name="InformationsUtilisateur" type="System.
        Configuration.DictionarySectionHandler" />
  </configSections>
  <InformationsUtilisateur>
    <add key="login" value="nico" />
    <add key="motdepasse" value="12345" />
    <add key="age" value="30" />
  </InformationsUtilisateur>
```

```
</configuration>
```

La première chose à voir est que nous indiquons à notre application que nous définissons une section de configuration du type `DictionarySectionHandler` et qui va s'appeler `InformationsUtilisateur`. Cela permet ensuite de définir notre propre section `InformationsUtilisateur`, qui ressemble beaucoup à la section `AppSettings` sauf qu'ici, on se rend tout de suite compte qu'il s'agit d'informations utilisateurs. Pour accéder à notre section depuis notre programme, nous devrons utiliser le code suivant :

```
Hashtable section = (Hashtable)ConfigurationManager.GetSection(
    "InformationsUtilisateur");
```

On utilise la méthode `GetSection` de la classe `ConfigurationManager` pour obtenir la section dont nous passons le nom en paramètre. On reçoit en retour une table de hachage (« `HashTable` ») et nous pourrons l'utiliser de cette façon pour obtenir les valeurs de nos clés :

```
Console.WriteLine(section["login"]);
Console.WriteLine(section["MOTDEPASSE"]);
Console.WriteLine(section["age"]);
```

Nous pouvons boucler sur les valeurs de la section en utilisant le code suivant :

```
foreach (DictionaryEntry d in section)
{
    Console.WriteLine("Clé  : " + d.Key + " / Valeur : " + d.
        Value);
}
```

Le type `NameValueSectionHandler`

La section de type `NameValueSectionHandler` ressemble beaucoup à la section précédente. Observons la configuration suivante :

```
<configuration>
  <configSections>
    <section name="InformationsUtilisateur" type="System.
        Configuration.NameValueSectionHandler" />
  </configSections>
  <InformationsUtilisateur>
    <add key="login" value="nico" />
    <add key="motdepasse" value="12345" />
    <add key="age" value="30" />
  </InformationsUtilisateur>
</configuration>
```

C'est la même que précédemment, à l'exception du type de la section, qui cette fois-ci est `NameValueSectionHandler`. Ce qui implique que nous obtenons un type différent en retour de l'appel à la méthode `GetSection`, à savoir un `NameValueCollection` :

```
NameValueCollection section = (NameValueCollection)
    ConfigurationManager.GetSection("InformationsUtilisateur");
```

La récupération des informations de configuration se fait de la même façon :

```
Console.WriteLine(section["login"]);
Console.WriteLine(section["MOTDEPASSE"]);
Console.WriteLine(section["age"]);
```

Ou encore avec une boucle pour toutes les récupérer :

```
foreach (string cle in section)
{
    Console.WriteLine("Clé  : " + cle + " / Valeur : " +
        section[cle]);
}
```

À vous de voir laquelle des deux méthodes vous préférez, mais dans tous les cas, il faudra fonctionner avec un système de clé / valeur.

Le type SingleTagSectionHandler

Ce troisième type permet de gérer une section différente des deux précédentes. Il sera possible d'avoir autant d'attributs que l'on souhaite dans la section. Prenez par exemple cette configuration :

```
<configuration>
  <configSections>
    <section name="MonUtilisateur" type="System.Configuration.
        SingleTagSectionHandler" />
  </configSections>
  <MonUtilisateur prenom="Nico" age="30" adresse="9 rue des
      bois"/>
</configuration>
```

Nous voyons que je peux mettre autant d'attributs que je le souhaite. Par contre, il ne sera possible de faire apparaître la section MonUtilisateur qu'une seule fois, alors que dans les sections précédentes, nous avions une liste de clé / valeur. Nous pourrons récupérer notre configuration avec le code suivant :

```
Hashtable section = (Hashtable)ConfigurationManager.GetSection(
    "MonUtilisateur");
Console.WriteLine(section["prenom"]);
Console.WriteLine(section["age"]);
Console.WriteLine(section["adresse"]);
```

Attention par contre, cette fois-ci la casse est importante pour obtenir la valeur de notre attribut. Notons notre boucle habituelle permettant de retrouver tous les attributs de notre section :

```
foreach (DictionaryEntry d in section)
{
    Console.WriteLine("Attribut  : " + d.Key + " / Valeur : " +
        d.Value);
}
```

Voilà pour les sections utilisant un type prédéfini.

Les groupes de sections

Super, nous savons définir des sections de configuration. Elles nous permettent d'organiser un peu mieux notre fichier de configuration. Par contre, si les sections se multiplient, cela va à nouveau être le bazar. Heureusement, les groupes de sections sont là pour remettre de l'ordre. Comme son nom l'indique, un groupe de sections va permettre de regrouper plusieurs sections. Le but est de clarifier le fichier de configuration. Regardons l'exemple suivant :

```
<configuration>
  <configSections>
    <sectionGroup name="Utilisateur">
      <section name="ParametreConnexion" type="System.
          Configuration.SingleTagSectionHandler" />
      <section name="InfoPersos" type="System.Configuration.
          DictionarySectionHandler" />
    </sectionGroup>
  </configSections>

  <Utilisateur>
    <ParametreConnexion Login="Nico" MotDePasse="12345" Mode="
        Authentification Locale"/>
    <InfoPersos>
      <add key="prenom" value="Nicolas" />
      <add key="age" value="30" />
    </InfoPersos>
  </Utilisateur>
</configuration>
```

Nous voyons ici que j'ai défini un groupe qui s'appelle `Utilisateur`, en utilisant la balise `sectionGroup`, contenant deux sections de configuration. Remarquons plus bas le contenu des sections et nous remarquons que la balise `<Utilisateur>` contient nos sections de configuration comme précédemment. Pour obtenir nos valeurs de configuration, la seule chose qui change est la façon de charger la section. Ici, nous mettons le nom de la section précédée du nom du groupe. Ce qui donne :

```
Hashtable section1 = (Hashtable)ConfigurationManager.GetSection
    ("Utilisateur/ParametreConnexion");
Hashtable section2 = (Hashtable)ConfigurationManager.GetSection
    ("Utilisateur/InfoPersos");
```

Après, la façon de récupérer les valeurs de configuration de chaque section reste la même. Avouez que c'est quand même plus clair non ?

Créer une section de configuration personnalisée

Nous allons étudier rapidement comment créer des sections de configuration personnalisées. Pour cela, il faut créer une section en dérivant de la classe `ConfigurationSection`. Cette classe permet de représenter une section d'un fichier de configuration. Donc, en toute logique, nous pouvons l'enrichir avec nos propriétés. Il suffit pour cela de décorer nos propres propriétés avec l'attribut `ConfigurationProperty`. Ce qui donne :

```
public class PersonneSection : ConfigurationSection
{
    [ConfigurationProperty("age", IsRequired = true)]
    public int Age
    {
        get { return (int)this["age"]; }
        set { this["age"] = value; }
    }

    [ConfigurationProperty("prenom", IsRequired = true)]
    public string Prenom
    {
        get { return (string)this["prenom"]; }
        set { this["prenom"] = value; }
    }
}
```

Le grand intérêt ici est de pouvoir typer les propriétés. Ainsi, nous pouvons avoir une section de configuration qui travaille avec un entier par exemple. Tout est fait par la classe mère ici et il suffit d'utiliser ses propriétés indexées en y accédant par son nom. Pour que notre section personnalisée soit reconnue, il faut la déclarer avec notre nouveau type :

```
<configSections>
  <section name="PersonneSection" type="MaPremiereApplication.
      PersonneSection, MaPremiereApplication" />
</configSections>
```

Le nom du type est constitué du nom complet du type (espace de noms + nom de la classe) suivi d'une virgule et du nom de l'assembly. Ici, l'espace de noms est le même que l'assembly, car j'ai créé mes classes à la racine du projet. Si vous avez un doute, vous devez vérifier l'espace de noms dans lequel est déclarée la classe. Ensuite, nous pourrons définir notre section, ce qui donne au final :

```
<configuration>
  <configSections>
    <section name="PersonneSection" type="
        MaPremiereApplication.PersonneSection,
```

```
        MaPremiereApplication " />
  </configSections>
  <PersonneSection prenom="nico" age="30"/>
</configuration>
```

Pour accéder aux informations contenues dans la section, il faudra charger la section comme d'habitude :

```
PersonneSection section = (PersonneSection)ConfigurationManager
    .GetSection("PersonneSection");
Console.WriteLine(section.Prenom + " a " + section.Age + " ans"
    );
```

Ce qui est intéressant de remarquer ici, c'est qu'on accède directement à nos propriétés via notre section personnalisée. Ce qui est une grande force et permet de travailler avec un entier et une chaîne de caractères. Il faudra faire attention à deux choses ici. La première est la casse. Comme on l'a vu dans le code, le nom est écrit en minuscule. Il faudra être cohérent entre le nom indiqué dans l'attribut `ConfigurationProperty`, celui indiqué en paramètre de l'opérateur d'indexation et celui écrit dans le fichier de configuration. Tout doit être orthographié de la même façon, sinon nous aurons une exception :

```
Exception non gérée : System.Configuration.
   ConfigurationErrorsException: Attribut 'PRENOM' non reconnu.
   Notez que les noms d'attributs respectent la casse. [.] \
   MaPremiereApplication\Program.cs:ligne 14
```

De même, si nous ne saisissons pas une valeur entière dans l'attribut `age`, il va y avoir un problème de conversion :

```
Exception non gérée : System.Configuration.
   ConfigurationErrorsException: La valeur de la propriété 'age'
    ne peut pas être analysée. L'erreur est : trente n'est pas
   une valeur valide pour Int32. [.] \MaPremiereApplication\
   Program.cs:ligne 14
```

Créer une section personnalisée avec une collection

Dans le paragraphe du dessus, on constate qu'on ne peut définir qu'un élément dans notre section. Il pourrait être intéressant dans certains cas d'avoir une section personnalisée qui puisse contenir plusieurs éléments, par exemple pour avoir une liste de personnes.

Pour ce faire, on utilisera la classe `ConfigurationPropertyCollection`. La première chose est de créer un élément en dérivant de la classe `ConfigurationElement`. Cet élément va ressembler beaucoup à ce qu'on a fait pour créer une section personnalisée :

```
public class ClientElement : ConfigurationElement
{
    private static readonly ConfigurationPropertyCollection _proprietes;
    private static readonly ConfigurationProperty age;
    private static readonly ConfigurationProperty prenom;

    static ClientElement()
    {
        prenom = new ConfigurationProperty("prenom", typeof(string), null, ConfigurationPropertyOptions.IsKey);
        age = new ConfigurationProperty("age", typeof(int), null, ConfigurationPropertyOptions.IsRequired);
        _proprietes = new ConfigurationPropertyCollection { prenom, age };
    }

    public string Prenom
    {
        get { return (string)this["prenom"]; }
        set { this["prenom"] = value; }
    }

    public int Age
    {
        get { return (int)this["age"]; }
        set { this["age"] = value; }
    }

    protected override ConfigurationPropertyCollection Properties
    {
        get { return _proprietes; }
    }
}
```

Ici, je définis deux propriétés, `Prenom` et `Age`, qui me permettent bien sûr d'y stocker un prénom et un âge qui sont respectivement une chaîne de caractères et un entier. À noter que nous avons besoin de décrire ces propriétés dans le constructeur statique de la classe. Pour cela, il faut lui indiquer son nom, c'est-à-dire la chaîne qui sera utilisée comme attribut dans l'élément de la section de configuration. Ensuite, nous lui indiquons son type ; pour cela on utilise le mot-clé `typeof` qui permet justement de renvoyer le type (dans le sens objet `Type`) d'un type. Enfin nous lui indiquons une option, par exemple le prénom sera la clé de mon élément (qui est une valeur unique et obligatoire à saisir) et l'âge, qui sera un élément obligatoire à saisir également. Ensuite, nous avons besoin d'utiliser cette classe à travers une collection d'éléments. Pour ce faire, il faut créer une classe qui dérive de `ConfigurationElementCollection` :

```
public class ClientElementCollection : ConfigurationElementCollection
```

```
{
    public override ConfigurationElementCollectionType
        CollectionType
    {
        get { return ConfigurationElementCollectionType.
            BasicMap; }
    }
    protected override string ElementName
    {
        get { return "Client"; }
    }

    protected override ConfigurationPropertyCollection
        Properties
    {
        get { return new ConfigurationPropertyCollection(); }
    }

    public ClientElement this[int index]
    {
        get { return (ClientElement)BaseGet(index); }
        set
        {
            if (BaseGet(index) != null)
            {
                BaseRemoveAt(index);
            }
            BaseAdd(index, value);
        }
    }

    public new ClientElement this[string nom]
    {
        get { return (ClientElement)BaseGet(nom); }
    }

    public void Add(ClientElement item)
    {
        BaseAdd(item);
    }

    public void Remove(ClientElement item)
    {
        BaseRemove(item);
    }

    public void RemoveAt(int index)
    {
        BaseRemoveAt(index);
    }
```

```

    public void Clear()
    {
        BaseClear();
    }

    protected override ConfigurationElement CreateNewElement()
    {
        return new ClientElement();
    }

    protected override object GetElementKey(ConfigurationElement element)
    {
        return ((ClientElement)element).Prenom;
    }
}
```

Vous pouvez copier ce code grâce au code web suivant :

▷ Copier ce code
Code web : 854375

Ces classes ont toujours la même structure. Ce qui est important de voir est qu'on utilise à l'intérieur la classe `ClientElement` pour indiquer le type de la collection. Nous indiquons également le nom de la balise qui sera utilisée dans le fichier de configuration, c'est la chaîne « Client » que renvoie la propriété `ElementName`. Enfin, j'ai la possibilité de définir ma clé en substituant la méthode `GetElementKey`. Le reste des méthodes appellent les méthodes de la classe mère.

Enfin, il faut créer notre section personnalisée, qui dérive comme d'habitude de `ConfigurationSection` :

```
public class ListeClientSection : ConfigurationSection
{
    private static readonly ConfigurationPropertyCollection proprietes;
    private static readonly ConfigurationProperty liste;

    static ListeClientSection()
    {
        liste = new ConfigurationProperty(string.Empty, typeof(ClientElementCollection), null, ConfigurationPropertyOptions.IsRequired | ConfigurationPropertyOptions.IsDefaultCollection);
        proprietes = new ConfigurationPropertyCollection { liste };
    }

    public ClientElementCollection Listes
    {
        get { return (ClientElementCollection)base[liste]; }
```

```
    }

    public new ClientElement this[string nom]
    {
        get { return Listes[nom]; }
    }

    protected override ConfigurationPropertyCollection
        Properties
    {
        get { return proprietes; }
    }
}
```

Notons dans cette classe comment nous utilisons l'opérateur d'indexation pour renvoyer un élément à partir de sa clé et renvoyer la liste des éléments. Maintenant, nous pouvons écrire notre configuration :

```
<configuration>
  <configSections>
    <section name="ListeClientSection" type="
        MaPremiereApplication.ListeClientSection,
        MaPremiereApplication" />
  </configSections>
  <ListeClientSection>
    <Client prenom="Nicolas" age="30"/>
    <Client prenom="Jérémie" age="20"/>
  </ListeClientSection>
</configuration>
```

Nous avons besoin à nouveau de définir la section en indiquant son type. Puis nous pouvons créer notre section et positionner notre liste de clients. Pour accéder à cette section, nous pouvons charger notre section comme avant avec la méthode `GetSection` :

```
ListeClientSection section = (ListeClientSection)
    ConfigurationManager.GetSection("ListeClientSection");
```

Puis nous pouvons itérer sur les éléments de notre section :

```
foreach (ClientElement clientElement in section.Listes)
{
    Console.WriteLine(clientElement.Prenom + " a " +
        clientElement.Age + " ans");
}
```

Ou bien, nous pouvons accéder à un élément à partir de sa clé :

```
ClientElement elementNicolas = section["Nicolas"];
Console.WriteLine(elementNicolas.Prenom + " a " +
    elementNicolas.Age + " ans");

ClientElement elementJeremie = section["Jérémie"];
```

```
Console.WriteLine(elementJeremie.Prenom + " a " +
    elementJeremie.Age + " ans");
```

Ce qui donnera :

```
Nicolas a 30 ans
Jérémie a 20 ans
```

Voilà pour la section personnalisée avec une liste d'éléments. Cette partie était peut-être un peu plus compliquée, mais retenons que la structure est souvent la même, il sera facile d'adapter vos sections à partir de ce code.

Nous avons découvert plusieurs façons de stocker des paramètres utilisables par notre application. Il faut savoir que beaucoup de composants du framework .NET sont intimement liés à ce fichier de configuration, comme une application web créée avec ASP.NET ou lorsqu'on utilise des services web.

Il est important de remarquer que ce fichier est un fichier de configuration d'application. Il y a d'autres endroits du même genre pour stocker de la configuration pour les applications .NET, comme le `machine.config` qui est un fichier de configuration partagé par toutes les applications de la machine. Il y a un héritage entre les différents fichiers de configuration. Si l'on définit une configuration au niveau machine (dans le `machine.config`), il est possible de la redéfinir pour notre application (`app.config`). En général, le fichier `machine.config` se trouve dans le répertoire d'installation du framework .NET, c'est-à-dire dans un sous répertoire du système d'exploitation, dépendant de la version du framework .NET installée. Chez moi par exemple, il se trouve dans le répertoire : `C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config`.

Enfin, remarquons qu'il est possible de faire des modifications du fichier de configuration directement depuis le code de notre application. C'est un point qui est rarement utilisé et que j'ai choisi de ne pas présenter pour que le chapitre ne soit pas trop long !

En résumé

- Les fichiers de configuration sont des fichiers XML qui possèdent les paramètres de configuration de notre application.
- Pour les applications console, ils s'appellent `app.config`.
- On peut définir toutes sortes de valeurs de configuration grâce aux sections prédéfinies ou en ajoutant son propre type de section personnalisée.

Chapitre 36

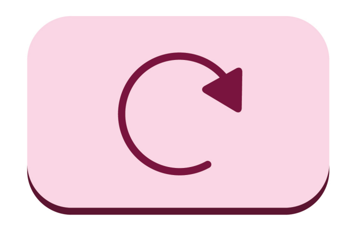
Introduction à LINQ

Difficulté : 

LINQ signifie *Language INtegrated Query*. C'est un ensemble d'extensions du langage permettant de faire des requêtes sur des données en faisant abstraction de leur type. Il permet d'utiliser facilement un jeu d'instructions supplémentaires afin de filtrer des données, faire des sélections, etc. Il existe plusieurs domaines d'applications pour LINQ :

– *Linq To Entities* ou *Linq To SQL* qui utilisent ces extensions de langage sur les bases de données.
– *Linq To XML* qui utilise ces extensions de langage pour travailler avec les fichiers XML.
– *Linq To Object* qui permet de travailler avec des collections d'objets en mémoire.

L'étude de LINQ nécessiterait un livre en entier, aussi nous allons nous concentrer sur la partie qui va le plus nous servir en tant que débutant et qui va nous permettre de commencer à travailler simplement avec cette nouvelle syntaxe, à savoir *Linq To Object*. Il s'agit d'extensions permettant de faire des requêtes sur les objets en mémoire et notamment sur toutes les listes ou collections. En fait, sur tout ce qui implémente `IEnumerable<>`.



Les requêtes Linq

Les requêtes Linq proposent une nouvelle syntaxe permettant d'écrire des requêtes qui ressemblent de loin à des requêtes SQL. Pour ceux qui ne connaissent pas le SQL, il s'agit d'un langage permettant de faire des requêtes sur les bases de données. Pour utiliser ces requêtes, il faut ajouter l'espace de noms adéquat, à savoir :

```
using System.Linq;
```

Ce `using` est en général inclus par défaut lorsqu'on crée un nouveau fichier.

Jusqu'à maintenant, si nous voulions afficher les entiers d'une liste d'entiers qui sont strictement supérieurs à 5, nous aurions fait :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
foreach (int i in liste)
{
    if (i > 5)
    {
        Console.WriteLine(i);
    }
}
```

Grâce à *Linq To Objet*, nous allons pouvoir filtrer en amont la liste afin de ne parcourir que les entiers qui nous intéressent, en faisant :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = from i in liste
                                  where i > 5
                                  select i;
foreach (int i in requeteFiltree)
{
    Console.WriteLine(i);
}
```

ce qui donnera :

```
6
9
15
8
```

Nous avons ici créé une requête Linq qui contient des mots-clés, comme `from`, `in`, `where` et `select`. Ici, cette requête veut dire que nous partons (`from`) de la liste « `liste` » en analysant chaque entier de la liste dans (`in`) la variable `i` où (`where`) `i` est supérieur à 5. Dans ce cas, l'entier est sélectionné comme faisant partie du résultat de la requête grâce au mot-clé `select`, qui fait un peu office de `return`. Cette requête renvoie un `IEnumerable<int>`. Le type générique est ici le type `int` car c'est le type de la variable `i` qui est retourné par le `select`. Le fait de renvoyer un `IEnumerable<>` va permettre potentiellement de réutiliser le résultat de cette requête pour un filtre successif ou une expression différente. En effet, Linq travaille sur des `IEnumerable<>`. Nous pourrions

par exemple ordonner cette liste par ordre croissant grâce au mot-clé `orderby`. Cela donnerait :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = from i in liste
                                  where i > 5
                                  select i;
IEnumerable<int> requeteOrdonnee = from i in requeteFiltree
                                   orderby i
                                   select i;
foreach (int i in requeteOrdonnee)
{
    Console.WriteLine(i);
}
```

qui pourrait également s'écrire en une seule fois avec :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requete = from i in liste
                           where i > 5
                           orderby i
                           select i;
foreach (int i in requete)
{
    Console.WriteLine(i);
}
```

Et nous aurons :

```
6
8
9
15
```

L'intérêt est que grâce à ces syntaxes, nous pouvons combiner facilement plusieurs filtres et construire des requêtes plus ou moins complexes. Par exemple, imaginons des clients :

```
public class Client
{
    public int Identifiant { get; set; }
    public string Nom { get; set; }
    public int Age { get; set; }
}
```

dont ou voudrait savoir s'ils sont majeurs, puis qu'on voudrait trier par `Age` puis par `Nom`, nous pourrions faire :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
```

```
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10}
};

IEnumerable<string> requete = from client in listeClients
                              where client.Age > 18
                              orderby client.Age, client.Nom
                              select client.Nom;

foreach (string prenom in requete)
{
    Console.WriteLine(prenom);
}
```

Ce qui donnera :

```
Jérémie
Delphine
Nicolas
```

Notez ici que mon `select` renvoie le nom du client, qui est un `string`. Il est donc normal que ma requête renvoie un `IEnumerable<string>` car j'ai choisi qu'elle ne sélectionne que les noms qui sont de type `string`. Il est assez fréquent de construire des objets anonymes à l'intérieur d'une requête. Par exemple, plutôt que de renvoyer uniquement le nom du client, je pourrais également renvoyer l'âge, mais comme je n'ai pas besoin de l'identifiant, je peux créer un objet anonyme juste avec ces deux propriétés :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10},
};

var requete = from client in listeClients
                              where client.Age > 18
                              orderby client.Age , client.Nom
                              select new { client.Nom, client
                                  .Age  };
foreach (var obj in requete)
{
    Console.WriteLine("{0} a {1} ans", obj.Nom, obj.Age);
}
```

Et nous aurons :

```
Jérémie a 20 ans
Delphine a 30 ans
Nicolas a 30 ans
```

Mon objet anonyme contient ici juste une propriété `Nom` et une propriété `Age`. À noter que je suis obligé à ce moment-là d'utiliser le mot-clé `var` pour définir la requête, car je n'ai pas de type à donner à cette requête. De même, dans le `foreach` je dois utiliser le mot-clé `var` pour définir le type anonyme. Les requêtes peuvent être de plus en plus compliquées, comme faisant des jointures. Par exemple, ajoutons une classe `Commande` :

```
public class Commande
{
    public int Identifiant { get; set; }
    public int IdentifiantClient { get; set; }
    public decimal Prix { get; set; }
}
```

Je peux créer des commandes pour des clients :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10},
};

List<Commande> listeCommandes = new List<Commande>
{
    new Commande{ Identifiant = 1, IdentifiantClient = 1, Prix
        = 150.05M},
    new Commande{ Identifiant = 2, IdentifiantClient = 2, Prix
        = 30M},
    new Commande{ Identifiant = 3, IdentifiantClient = 1, Prix
        = 99.99M},
    new Commande{ Identifiant = 4, IdentifiantClient = 1, Prix
        = 100M},
    new Commande{ Identifiant = 5, IdentifiantClient = 3, Prix
        = 80M},
    new Commande{ Identifiant = 6, IdentifiantClient = 3, Prix
        = 10M},
};
```

Et grâce à une jointure, récupérer avec ma requête le nom du client et le prix de sa commande :

```
var liste = from commande in listeCommandes
            join client in listeClients on commande.
                IdentifiantClient equals client.Identifiant
            select new { client.Nom, commande.Prix };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} a payé {1}", element.Nom,
        element.Prix);
}
```

Ce qui donne :

```
Le client Nicolas a payé 150,05
Le client Jérémie a payé 30
Le client Nicolas a payé 99,99
Le client Nicolas a payé 100
Le client Delphine a payé 80
Le client Delphine a payé 10
```

On utilise le mot-clé `join` pour faire la jointure entre les deux listes puis on utilise le mot-clé `on` et le mot-clé `equals` pour indiquer sur quoi on fait la jointure. À noter que ceci peut se réaliser en imbriquant également les `from` et en filtrant sur l'égalité des identifiants clients :

```
var liste = from commande in listeCommandes
            from client in listeClients
            where client.Identifiant == commande.
                IdentifiantClient
            select new { client.Nom, commande.Prix };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} a payé {1}", element.Nom,
        element.Prix);
}
```

Il est intéressant de pouvoir regrouper les objets qui ont la même valeur. Par exemple pour obtenir toutes les commandes, groupées par client, on fera :

```
var liste = from commande in listeCommandes
            group commande by commande.IdentifiantClient;

foreach (var element in liste)
{
    Console.WriteLine("Le client : {0} a réalisé {1} commande(s
        )", element.Key, element.Count());
    foreach (Commande commande in element)
    {
        Console.WriteLine("\tPrix : {0}", commande.Prix);
    }
}
```

Ici, cela donne :

```
Le client : 1 a réalisé 3 commande(s)
        Prix : 150,05
        Prix : 99,99
        Prix : 100
Le client : 2 a réalisé 1 commande(s)
        Prix : 30
Le client : 3 a réalisé 2 commande(s)
```

```
        Prix : 80
        Prix : 10
```

Il est possible de cumuler le `group by` avec notre jointure précédente histoire d'avoir également le nom du client :

```
var liste = from commande in listeCommandes
            join client in listeClients on commande.
                IdentifiantClient equals client.Identifiant
            group commande by new {commande.IdentifiantClient,
                client.Nom};

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2}
        commande(s)", element.Key.Nom, element.Key.
        IdentifiantClient, element.Count());
    foreach (Commande commande in element)
    {
        Console.WriteLine("\tPrix : {0}", commande.Prix);
    }
}
```

Et nous obtenons :

```
Le client Nicolas (1) a réalisé 3 commande(s)
        Prix : 150,05
        Prix : 99,99
        Prix : 100
Le client Jérémie (2) a réalisé 1 commande(s)
        Prix : 30
Le client Delphine (3) a réalisé 2 commande(s)
        Prix : 80
        Prix : 10
```

À noter que le `group by` termine la requête, un peu comme le `select`. Ainsi, si l'on veut sélectionner quelque chose après le `group by`, il faudra utiliser le mot-clé `into` et la syntaxe suivante :

```
var liste = from commande in listeCommandes
            join client in listeClients on commande.
                IdentifiantClient equals client.Identifiant
            group commande by new {commande.IdentifiantClient,
                client.Nom} into commandesGroupees
            select
                new
                {
                    commandesGroupees.Key.IdentifiantClient,
                    commandesGroupees.Key.Nom,
                    NombreDeCommandes = commandesGroupees.Count
                        ()
```

```
                    };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2} commande(s)", element.Nom, element.IdentifiantClient, element.NombreDeCommandes);
}
```

Avec pour résultat :

```
Le client Nicolas (1) a réalisé 3 commande(s)
Le client Jérémie (2) a réalisé 1 commande(s)
Le client Delphine (3) a réalisé 2 commande(s)
```

L'intérêt d'utiliser le mot-clé `into` est également de pouvoir enchaîner avec une autre jointure ou un autre filtre permettant de continuer la requête. Il est également possible d'utiliser des variables à l'intérieur des requêtes grâce au mot-clé `let`. Cela va nous permettre de stocker des résultats temporaires pour les réutiliser ensuite :

```
var liste = from commande in listeCommandes
            join client in listeClients on commande.IdentifiantClient equals client.Identifiant
            group commande by new {commande.IdentifiantClient, client.Nom} into commandesGroupees
            let total = commandesGroupees.Sum(c => c.Prix)
            where total > 50
            orderby total
            select new
                {
                    commandesGroupees.Key.IdentifiantClient,
                    commandesGroupees.Key.Nom,
                    NombreDeCommandes = commandesGroupees.Count(),
                    PrixTotal = total
                };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2} commande(s) pour un total de {3}", element.Nom, element.IdentifiantClient, element.NombreDeCommandes, element.PrixTotal);
}
```

Par exemple, ici j'utilise le mot-clé `let` pour stocker le total d'une commande groupée dans la variable `total` (nous verrons la méthode `Sum()` un tout petit peu plus bas), ce qui me permet ensuite de filtrer avec un `where` pour obtenir les commandes dont le total est supérieur à 50 et de les trier par ordre de prix croissant.

Ce qui donne :

```
Le client Delphine (3) a réalisé 2 commande(s) pour un total de
    90
Le client Nicolas (1) a réalisé 3 commande(s) pour un total de
    350,04
```

Nous allons nous arrêter là pour cet aperçu des requêtes LINQ. Nous avons pu voir que le C# dispose d'un certain nombre de mots-clés qui permettent de manipuler nos données de manière très puissante mais d'une façon un peu inhabituelle.



Cette façon d'écrire des requêtes LINQ s'appelle en anglais la *sugar syntax*, que l'on peut traduire par « sucre syntaxique ». Il désigne de manière générale les constructions d'un langage qui facilitent la rédaction du code sans modifier l'expressivité du langage.

Voyons à présent ce qu'il y a derrière cette jolie syntaxe.

Les méthodes d'extension Linq

En fait, toute la *sugar syntax* que nous avons vue précédemment repose sur un certain nombre de méthodes d'extension qui travaillent sur les types `IEnumerable<T>`. Par exemple, la requête suivante :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = from i in liste
                                  where i > 5
                                  select i;
foreach (int i in requeteFiltree)
{
    Console.WriteLine(i);
}
```

s'écrit véritablement :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = liste.Where(i => i > 5);
foreach (int i in requeteFiltree)
{
    Console.WriteLine(i);
}
```

Nous utilisons la méthode d'extension `Where()` en lui fournissant une expression lambda servant de prédicat pour filtrer la liste. C'est de cette façon que le compilateur traduit la *sugar syntax*. Ce n'est donc qu'une façon plus élégante d'utiliser ces méthodes d'extension. Chaque méthode d'extension renvoie un `IEnumerable<T>` ce qui permet d'enchaîner facilement les filtres successifs. Par exemple, rajoutons une date et un nombre d'articles à notre classe `Commande` :

```
public class Commande
{
    public int Identifiant { get; set; }
    public int IdentifiantClient { get; set; }
    public decimal Prix { get; set; }
    public DateTime Date { get; set; }
    public int NombreArticles { get; set; }
 }
```

Avec la requête suivante :

```
IEnumerable<Commande> commandesFiltrees = listeCommandes.
Where(commande => commande.Prix > 100).
Where(commande => commande.NombreArticles > 10).
OrderBy(commande => commande.Prix).
ThenBy(commande => commande.DateAchat);
```

nous pouvons obtenir les commandes dont le prix est supérieur à 100, où le nombre d'articles est supérieur à 10, triées par prix puis par date d'achat.

De plus, ces méthodes d'extension font beaucoup plus de choses que ce que l'on peut faire avec la *sugar syntax*. Il existe pas mal de méthodes intéressantes, que nous ne pourrons pas toutes étudier. Regardons par exemple la méthode `Sum()` (qui a été utilisée dans le paragraphe précédent) qui permet de faire la somme des éléments d'une liste ou la méthode `Average()` qui permet d'en faire la moyenne :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
Console.WriteLine("Somme : {0}", liste.Sum());
Console.WriteLine("Moyenne : {0}", liste.Average());
```

Qui nous renvoie dans ce cas :

```
Somme : 51
Moyenne : 6,375
```

Tout est déjà fait, c'est pratique ! Évidemment, les surcharges de ces deux méthodes d'extension ne fonctionnent qu'avec des types `int` ou `double` ou `decimal`... Qui envisagerait de faire une moyenne sur une chaîne ? Par contre, il est possible de définir une expression lambda dans la méthode `Sum()` afin de faire la somme sur un élément d'un objet, comme le prix de notre commande :

```
decimal prixTotal = listeCommandes.Sum(commande => commande.
    Prix);
```

D'autres méthodes sont bien utiles. Par exemple la méthode d'extension `Take()` nous permet de récupérer les N premiers éléments d'une liste :

```
IEnumerable<Client> extrait = listeClients.OrderByDescending(
    client => client.Age).Take(5);
```

Ici, je trie dans un premier temps ma liste par âge décroissant, et je prends les 5 premiers. Ce qui signifie que je prends les 5 plus vieux clients de ma liste. Et s'il n'y

en a que 3 ? eh bien il prendra uniquement les 3 premiers ! Suivant le même principe, on peut utiliser la méthode `First()` pour obtenir le premier élément d'une liste :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
int premier = liste.Where(i => i > 5).First();
```

J'obtiens alors le premier élément de la liste qui est strictement supérieur à 5. À noter que le filtre peut également se faire dans l'expression lambda de la méthode `First()` :

```
int premier = liste.First(i => i > 5);
```

Ce qui revient exactement au même. Attention, s'il n'y a aucun élément dans la liste, alors la méthode `First()` lève l'exception :

```
Exception non gérée : System.InvalidOperationException: La sé
   quence ne contient aucun élément.
```

Il est possible dans ce cas-là d'éviter une exception avec la méthode `FirstOrDefault()` qui renvoie la valeur par défaut du type de la liste (0 si c'est un type valeur, `null` si c'est un type référence) :

```
Client nicolas = listeClients.FirstOrDefault(client => client.
    Nom == "Nicolas");
if (nicolas == null)
     Console.WriteLine("Client non trouvé");
```

Ici, je cherche le premier des clients dont le nom est Nicolas. S'il n'est pas trouvé, alors `FirstOrDefault()` me renvoie `null`, sinon, il me renvoie bien sûr le bon objet `Client`. Dans le même genre, nous pouvons compter grâce à la méthode `Count()` le nombre d'éléments d'une source de données suivant un critère :

```
int nombreClientsMajeurs = listeClients.Count(client => client.
    Age >= 18);
```

Ici, j'obtiendrai le nombre de clients majeurs dans ma liste. De la même façon qu'avec la *sugar syntax*, il est possible de faire une sélection précise des données que l'on souhaite extraire, grâce à la méthode `Select()` :

```
var requete = listeClients.Where(client => client.Age >= 18).
    Select(client => new { client.Age, client.Nom }) ;
```

Cela me permettra d'obtenir une requête contenant les clients majeurs. À noter que seront retournés des objets anonymes possédant une propriété `Age` et une propriété `Nom`. Bien sûr, nous retrouverons nos jointures avec la méthode d'extension `Join()` ou les groupes avec la méthode `GroupBy()`. Il existe beaucoup de méthodes d'extension et il n'est pas envisageable dans ce livre de toutes les décrire. Je vais finir en vous parlant des méthodes `ToList()` et `ToArray()` qui, comme leurs noms le suggèrent, permettent de forcer la requête à être mise dans une liste ou dans un tableau :

```
List<Client> lesPlusVieuxClients = listeClients.
    OrderByDescending(client => client.Age).Take(5).ToList();
```

ou

```
Client[] lesPlusVieuxClients = listeClients.OrderByDescending(
    client => client.Age).Take(5).ToArray();
```

Plutôt que d'avoir un `IEnumerable<>`, nous obtiendrons cette fois-ci une `List<>` ou un tableau. Le fait d'utiliser ces méthodes d'extension a des conséquences que nous allons décrire.

Exécution différée

Les méthodes d'extension LINQ ou sa syntaxe sucrée c'est bien joli, mais quel est l'intérêt de s'en servir plutôt que d'utiliser des boucles `foreach`, des `if` ou d'autres choses ? Déjà, parce qu'il y a plein de choses déjà toutes faites : la somme, la moyenne, la récupération de N éléments, etc. Mais aussi pour une autre raison plus importante : **l'exécution différée**.

Nous en avons déjà parlé, l'exécution différée est possible grâce au mot-clé `yield`. Les méthodes d'extensions Linq utilisent fortement ce principe. Cela veut dire que lorsque nous construisons une requête, elle n'est pas exécutée tant que l'on n'itère pas sur le contenu de la requête. Ceci permet de stocker la requête, d'empiler éventuellement des filtres ou des jointures et de ne pas calculer le résultat tant qu'on n'en a pas explicitement besoin.

Ainsi, imaginons que nous souhaitions trier une liste d'entiers. Avant cela, nous aurions fait :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };

liste.Sort();
liste.Add(7);

foreach (int i in liste)
{
    Console.WriteLine(i);
}
```

Ce qui aurait affiché en toute logique la liste triée puis, à la fin, l'entier 7 rajouté, c'est-à-dire :

```
1
3
4
5
6
8
9
15
7
```

Avec Linq, nous allons pouvoir écrire :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };

var requete = liste.OrderBy(e => e);
liste.Add(7);

foreach (int i in requete)
{
    Console.WriteLine(i);
}
```

Et si nous exécutons ce code, nous aurons :

```
1
3
4
5
6
7
8
9
15
```

Bien que nous ayons ajouté la valeur 7 après avoir trié la liste avec `OrderBy`, on se rend compte que tous les entiers sont quand même triés lorsque nous les affichons.

En effet, la requête n'a été exécutée qu'au moment du `foreach`. Ceci implique donc que le tri va tenir compte de l'ajout du 7 à la liste. La requête est construite en mémorisant les conditions comme notre `OrderBy`, mais cela fonctionne également avec un `where`, et tout ceci n'est exécuté que lorsqu'on le demande explicitement ; c'est-à-dire avec un `foreach` dans ce cas-là. En fait, tant que le C# n'est pas obligé de parcourir les éléments énumérables alors il ne le fait pas. Ce qui permet d'enchaîner les éventuelles conditions et d'éviter les parcours inutiles. Par exemple, dans le cas ci-dessous, il est inutile d'exécuter le premier filtre :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requete = liste.Where(i => i > 5);
// plein de choses qui n'ont rien à voir avec la requete
requete = requete.Where(i => i > 10);
```

En effet, le deuxième filtre a tout intérêt à être combiné au premier afin d'être simplifié. Et encore, ici, on n'utilise même pas la requête, il y a encore moins d'intérêt à effectuer nos filtres si nous ne nous servons pas du résultat. Ceci peut paraître inattendu, mais c'est très important dans la façon dont Linq s'en sert afin d'optimiser ses requêtes. Ici, le parcours en mémoire pourrait paraître peu coûteux, mais dans la mesure où Linq doit fonctionner aussi bien avec des objets, qu'avec des bases de données ou du XML (ou autres. . .), cette optimisation prend tout son sens.

Le maître mot est la **performance**, primordial quand on accède aux bases de données. Cette exécution différée est gardée pour le plus tard possible. C'est-à-dire que le fait

de parcourir notre boucle va obligatoirement entraîner l'évaluation de la requête afin de pouvoir retourner les résultats cohérents. Il en va de même pour certaines autres opérations, comme la méthode `Sum()`. Comment pourrions-nous faire la somme de tous les éléments si nous ne les parcourons pas ? C'est aussi le cas pour les méthodes `ToList()` et `ToArray()`. Par contre, ce n'est pas le cas pour les méthodes `Where`, ou `Take`, etc.

Il est important de connaître ce mécanisme. L'exécution différée est très puissante et connaître son fonctionnement permet de savoir exactement ce que nous faisons et pourquoi nous pouvons obtenir parfois des résultats étranges.

Récapitulatif des opérateurs de requêtes

Pour terminer avec Linq, voici un tableau récapitulatif des différents opérateurs de requête. Nous ne les avons pas tous étudiés ici car cela serait bien vite lassant. Mais grâce à leurs noms et leurs types, il est assez facile de voir à quoi ils servent afin de les utiliser dans la construction de nos requêtes.

Type	**Opérateur de requête**	**Exécution différée**
Tri des données	`OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `Reverse`	Oui
Opérations ensemblistes	`Distinct`, `Except`, `Intersect`, `Union`	Oui
Filtrage des données	`OfType`, `Where`	Oui
Opérations de quantificateur	`All`, `Any`, `Contains`	Non
Opérations de projection	`Select`, `SelectMany`	Oui
Partitionnement des données	`Skip`, `SkipWhile`, `Take`, `TakeWhile`	Oui
Opérations de jointure	`Join`, `GroupJoin`	Oui
Regroupement de données	`GroupBy`, `ToLookup`	Oui
Opérations de génération	`DefaultIfEmpty`, `Empty`, `Range`, `Repeat`	Oui
Opérations d'égalité	`SequenceEqual`	Non
Opérations d'élément	`ElementAt`, `ElementAtOrDefault`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`	Non
Conversion de types de données	`AsEnumerable`, `AsQueryable`, `Cast`, `OfType`, `ToArray`, `ToDictionary`, `ToList`, `ToLookup`	Non
Opérations de concaténation	`Concat`	Oui
Opérations d'agrégation	`Aggregate`, `Average`, `Count`, `LongCount`, `Max`, `Min`, `Sum`	Non

N'hésitez pas à consulter la documentation de ces méthodes d'extension ou à aller voir des exemples sur internet. Il y a beaucoup de choses à faire avec ces méthodes. Il est important également de bien savoir les maîtriser afin d'éviter les problèmes de performance. En effet, l'évaluation systématique des expressions peut être coûteuse, surtout quand le tout est imbriqué dans des boucles. À utiliser judicieusement !

Voilà pour ce petit aperçu de Linq. Rappelez-vous bien que Linq est une abstraction qui permet de manipuler des sources de données différentes. Nous avons vu son utilisation avec les objets implémentant `IEnumerable<>`, avec ce qu'on appelle *Linq To Objects*. Il est possible de faire du Linq en allant manipuler des données en base de données, on utilisera pour cela *Linq To SQL* ou *Linq To Entity*. De même, il est possible de manipuler les fichiers XML avec *Linq To XML*.

Linq apporte des méthodes d'extension et une syntaxe complémentaire afin d'être efficace avec la manipulation de sources de données.

Sachez enfin qu'il est possible de requêter n'importe quelle source de données à partir du moment où un connecteur spécifique a été développé. Cela a été fait par exemple pour interroger Google ou Amazon, mais aussi pour requêter sur active directory, ou JSON, etc.

En résumé

- Linq consiste en un ensemble d'extensions du langage permettant de faire des requêtes sur des données en faisant abstraction de leur type.
- Il existe plusieurs domaines d'applications de Linq, comme *Linq to Object*, *Linq to Sql*, etc.
- La *sugar syntax* ajoute des mots-clés qui permettent de faire des requêtes qui ressemblent aux requêtes faites avec le langage SQL.
- Derrière cette syntaxe se cache un bon nombre de méthodes d'extension qui tirent parti des mécanismes d'exécution différée.

Chapitre 37

Accéder aux données avec Entity Framework

Difficulté : 

Nous allons voir dans ce chapitre comment connecter nos applications à une base de données. Bien qu'il soit possible de travailler avec quasiment n'importe quelle base de données (oracle, mysql,...), le framework .NET offre toute sa puissance en fonctionnant avec SQL Server.

Si vous vous souvenez, lors de l'installation de Visual C# Express, nous avons également installé Microsoft SQL Server 2008 express Service Pack 1. SQL Server 2008 Express est un moteur de base de données *light*, idéal pour travailler en local sur son PC. Dans un environnement de production, nous aurons tout intérêt à travailler avec la version complète de SQL Server, mais pour ce livre, la version express est amplement suffisante. En plus... elle est gratuite !



Les bases de données et la modélisation

Une base de données est un gros espace de stockage structuré qui permet d'enregistrer efficacement de très grandes quantités d'informations. C'est un élément incontournable dans tout système informatique. Dans une base de données, on peut stocker des informations unitaires, comme le nom d'un client, son âge, son adresse, etc. Ce sont les **champs**. Ces champs sont regroupés sémantiquement dans des **tables**. Par exemple, la table des clients contient les informations relatives au client Nicolas, au client Jérémie, etc. Ces informations sont appelées des enregistrements.

Lorsqu'on représentera les données de notre application, nous allons avoir besoin de plusieurs tables. Par exemple, pour représenter les données d'une application de commerce, nous pouvons avoir une table « Produits », une table « Rayons », une table « Clients », une table « Commandes », etc. Il peut y avoir des relations entre les tables, par exemple un rayon peut contenir de 0 à N produits, un client peut passer de 0 à N commandes, etc.

On appelle ces bases de données des **bases de données relationnelles**.

Nous pouvons lire le contenu des tables et insérer de nouvelles valeurs grâce au langage SQL. Nous allons très peu nous en servir mais il est la base de tout requêtage en base de données. Nous n'allons pas faire ici de cours sur le SQL mais nous allons l'utiliser à certains endroits, par souci de simplicité.

La première chose à faire est de réfléchir au modèle de données dont on aura besoin dans notre application. Pour ce chapitre, nous allons prendre pour exemple une application de commerce, dans le genre site d'e-commerce, spécialisée dans la création d'applications console. Nous allons commencer par modéliser les rayons et les produits. On a dit qu'un rayon pouvait être composé de 0 à N produits. De même, un produit peut appartenir à 0 ou à N rayons. Un rayon possède un identifiant, un nom et une description. Un produit possède un identifiant, un nom, un prix, un stock et une url vers son image (voir la figure 37.1).

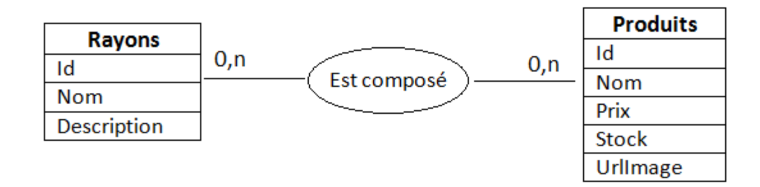


FIGURE 37.1 – On appelle ce schéma un modèle conceptuel de données

Entity Framework et le mapping objet relationnel

Finalement, la modélisation que nous venons de faire ressemble beaucoup à la modélisation orientée objet que nous commençons à maîtriser. On pourrait très bien avoir des

objets **Produit**, des objets **Rayon**. Un rayon pourrait contenir des produits, etc. Ne pourrions-nous pas essayer de représenter la base de données sous la forme d'un modèle orienté objet plutôt qu'un modèle relationnel ? C'est le principe de ce que l'on appelle un **ORM**[1] que l'on traduit en français par « mapping objet-relationnel ». L'ORM est un outil qui permet de générer une couche d'accès aux données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et des objets. Il existe plusieurs outils d'ORM pour .NET permettant de générer des objets C#, comme nhibernate, Entity Framework, etc.

Nous allons utiliser ici Entity Framework. C'est l'ORM de Microsoft. Il est totalement intégré à Visual C# Express et aux autres versions. Son travail consiste, entre autres, à :

– modéliser ses données et générer la base correspondante ;
– générer un modèle à partir d'une base de données existante ;
– gérer tous les accès à la base de données (lecture, écriture, suppression,. . .).

Le grand intérêt est que nous allons travailler directement avec des objets et c'est lui qui s'occupera de tout ce qui est persistance dans la base de données.

Allez, fini le blabla, passons à son utilisation ! Commençons par créer une application console `MonApplicationBaseDeDonnées`, ainsi qu'indiqué à la figure 37.2.

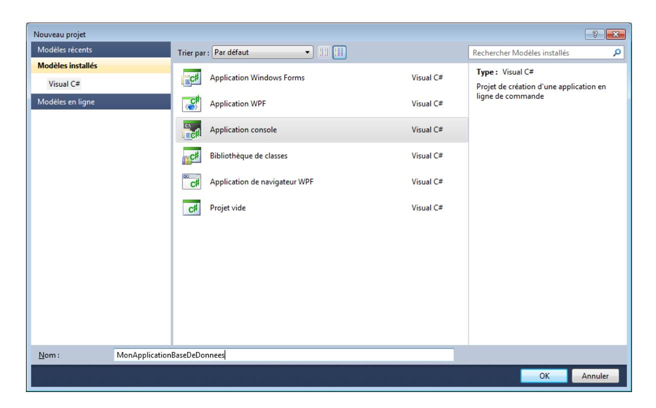


FIGURE 37.2 – Création de l'application console

À présent, ajoutons un nouvel élément de type `ADO.NET Entity Data Model` à notre projet, que l'on va appeler `ModelCommerce.edmx` (voir figure 37.3).

C'est ce type de fichier qui va nous permettre de modéliser nos données. L'assistant s'ouvre et nous choisissons `Modèle vide`, comme à la figure 37.4.

On voit apparaître une nouvelle fenêtre intitulée `Entity Data Model Designer`. Cette fenêtre « designer » va nous permettre de modéliser nos données. Elle nous donne également accès à la boîte à outils, ainsi que vous pouvez le voir à la figure 37.5.

Lorsque nous accédons à la boîte à outils, nous pouvons voir plusieurs éléments. Celui qui nous intéresse est l'entité (voir figure 37.6).

1. C'est l'acronyme de l'expression anglaise : *object relationnal mapping*.

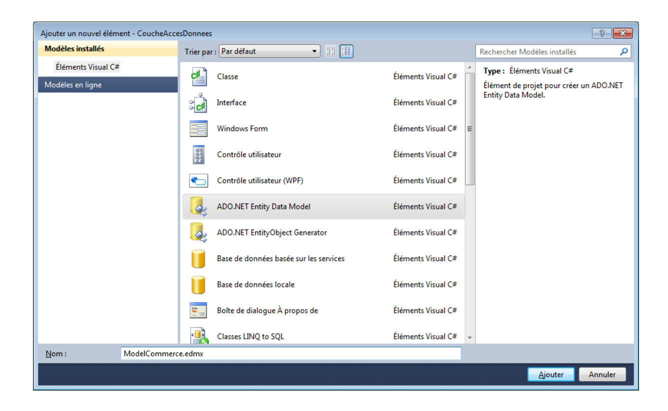


Figure 37.3 – Ajout d'un fichier ADO.NET Entity Data Model

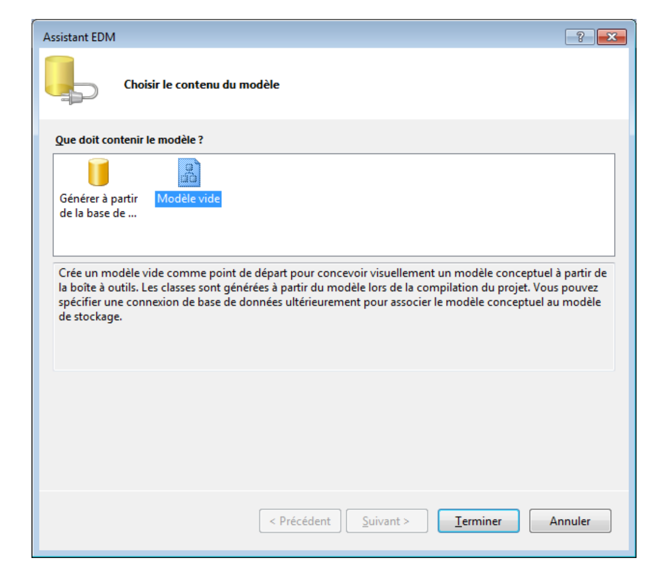


Figure 37.4 – Choix d'un modèle vide

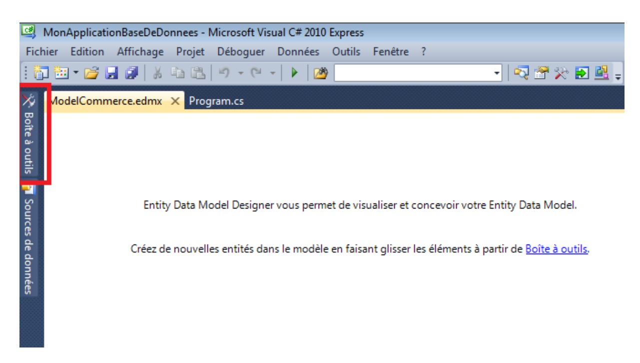


Figure 37.5 – La boîte à outil est accessible sur la gauche

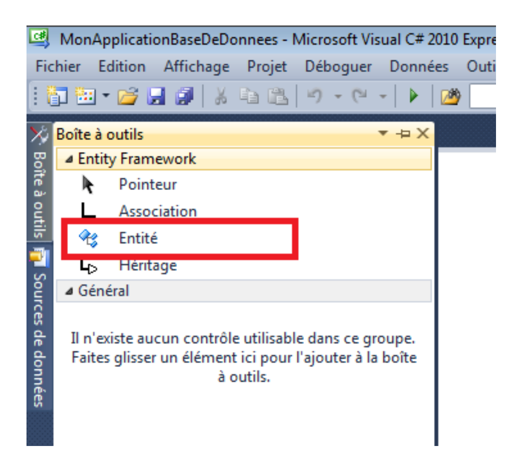


Figure 37.6 – Ajout d'une entité sur le designer

Nous allons pouvoir modéliser nos entités en les faisant glisser sur le designer. Comme l'illustre la figure 37.7, nous la voyons apparaître sur le designer.



Figure 37.7 – L'entité apparaît sur le designer

Nous pouvons renommer cette entité, soit en allant modifier la propriété `Nom` dans la fenêtre de propriétés, soit en cliquant directement sur le nom dans le designer. Appelons cette entité `Rayon`. Nous pouvons constater que cette entité est générée avec une propriété par défaut : `ID`. C'est l'identifiant de notre rayon. Cet identifiant possède des propriétés que nous pouvons voir dans la fenêtre de propriétés (voir figure 37.8).

Nous pouvons par exemple voir (ou modifier) son nom (ID) et son type (`Int32`, qui est l'équivalent de `int`). Nous voyons également que cette propriété est la clé d'entité. Ce qui veut dire que c'est ce qui va nous permettre d'identifier notre rayon de manière unique (c'est un peu plus complexe que ça, mais retenons ce point). Nous pouvons ajouter une nouvelle propriété à l'entité ; pour cela il suffit de faire un clic droit sur l'entité et d'ajouter une propriété scalaire (voir figure 37.9).

Une nouvelle propriété apparaît dans notre entité `Rayon` (voir figure 37.10).

Nous pouvons la renommer en `Nom`. Nous pouvons voir sur la figure 37.11 qu'elle est par défaut du type `String`, ce qui nous va très bien.

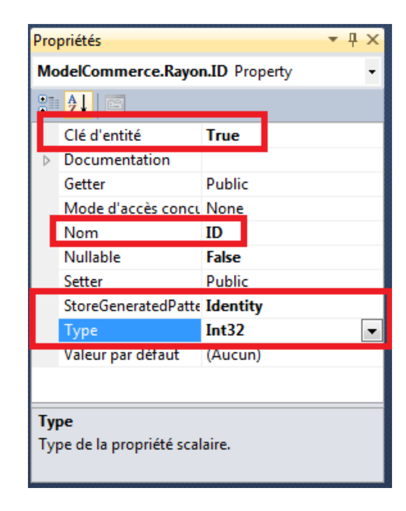


FIGURE 37.8 – Les propriétés de la propriété ID

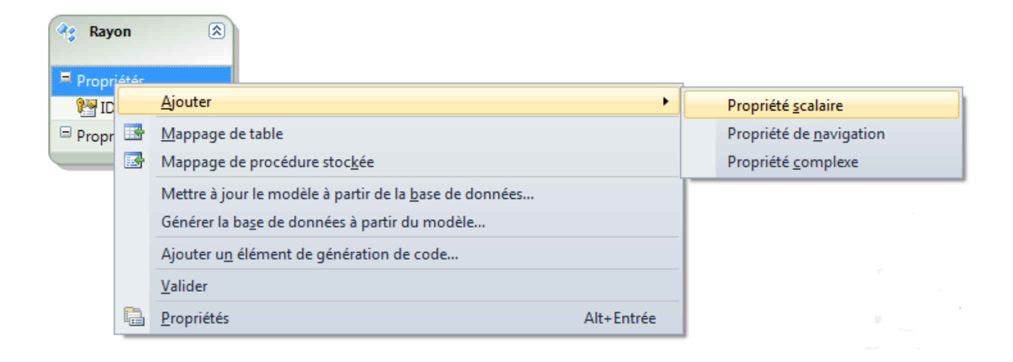


FIGURE 37.9 – Ajout d'une propriété scalaire

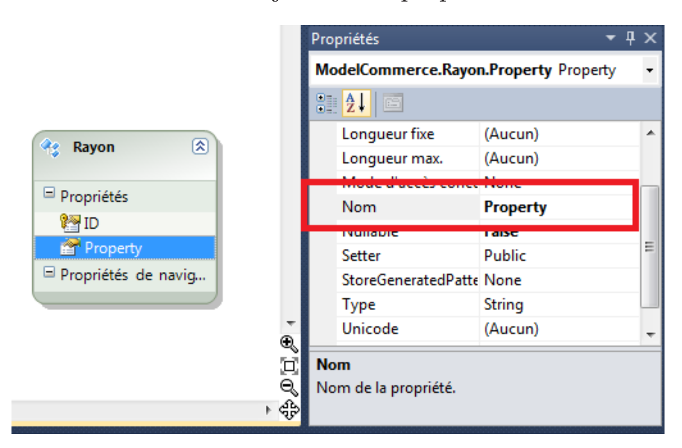


FIGURE 37.10 – Changement du nom de la propriété

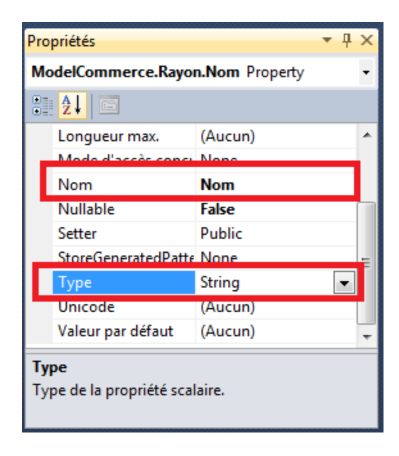
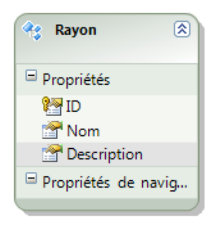


FIGURE 37.11 – Le nom et le type de la propriété

Rajoutons une troisième propriété `Description`, toujours de type `String` mais qui pourra être nulle, il suffit de déclarer la propriété Nullable à `True` (voir la figure 37.12).



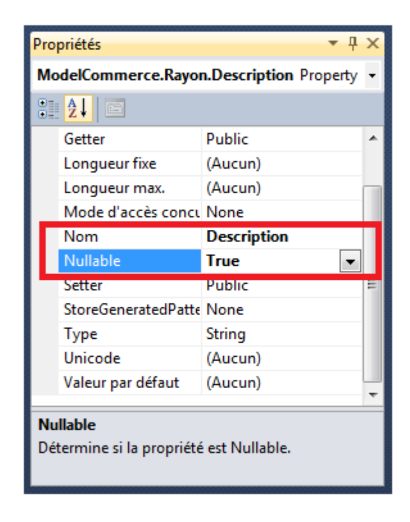


FIGURE 37.12 – La propriété peut être nulle

Ça y est, notre entité `Rayon` est modélisée ! Rajoutons maintenant une nouvelle entité, `Produit`, qui possède également un identifiant, un nom de type `String` et un prix de type `Decimal`. Pour mettre le type à `Decimal`, il suffit de changer le type dans la fenêtre des propriétés (voir la figure 37.13).

Ensuite, rajoutons une propriété `Stock` de type `Int32` et une propriété `UrlImage` de type `String`. Nous obtenons deux superbes entités (voir la figure 37.14).

Il est temps de relier les entités entre elles grâce à une association : faites un clic droit sur l'entité `Rayon` et ajoutez une association, comme indiqué à la figure 37.15.

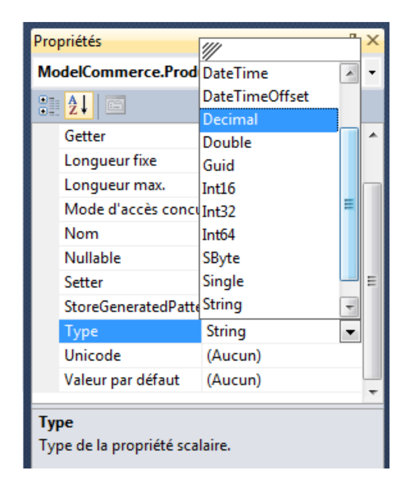


FIGURE 37.13 – Changement du type de la propriété

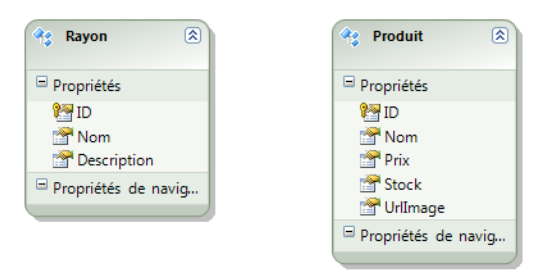


FIGURE 37.14 – Les deux entités de notre modèle

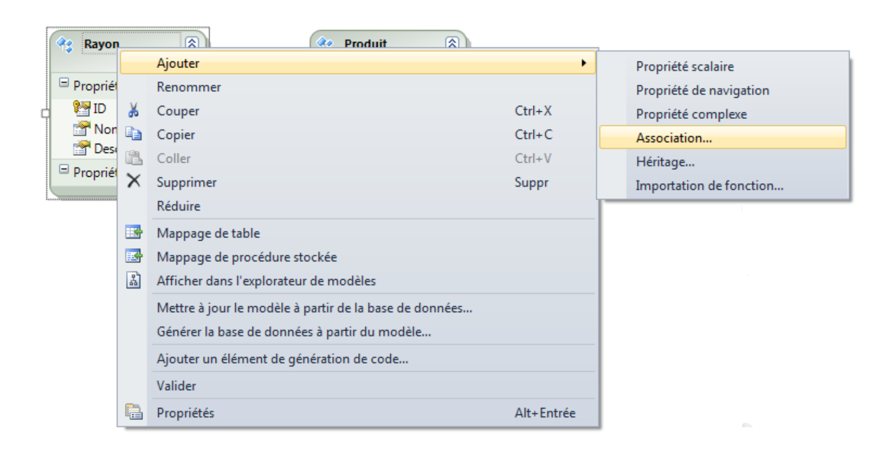


FIGURE 37.15 – Ajout d'une association entre les deux entités

Un nouvel écran s'ouvre qui permet de définir l'association. Indiquons que la multiplicité est à « plusieurs » sur les deux entités, ce qui permet de dire qu'un rayon peut contenir de 0 à N produits et inversement, un produit peut être contenu dans 0 à N rayons. Notons au passage que les choix possibles sont 1, 0 ou 1 et plusieurs. C'est ce qui nous permet d'indiquer la cardinalité de nos relations. Changez ensuite le nom des propriétés de navigation en rajoutant un « s » à `Produit` et à `Rayon`, comme indiqué à la figure 37.16.

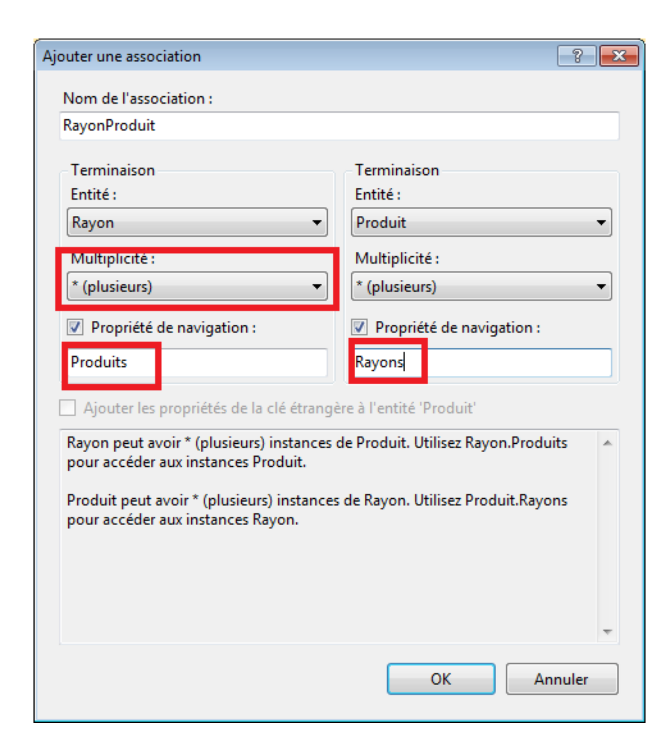


FIGURE 37.16 – Changement des caractéristiques de l'association

Le designer est mis à jour avec la relation et on peut voir apparaître des propriétés de navigation dans les entités. L'entité `Rayon` a une propriété `Produits`, ce qui va permettre d'obtenir la liste des produits d'un `Rayon`. De même, l'entité `Produit` possède une propriété de navigation `Rayons` qui va permettre d'obtenir la liste des rayons qui contiennent le produit (voir la figure 37.17).

Nous allons encore faire une petite modification à ce modèle. Sélectionnez l'entité `Rayon`. Nous pouvons voir dans ses propriétés que le nom du jeu d'entité vaut `RayonJeu`, modifiez-le en `Rayons`, comme indiqué à la figure 37.18.

Faites pareil pour l'entité `Produit` : changez `ProduitJeu` en `Produits`.

Voilà, tout ça, c'est notre modèle ! Il faut maintenant faire en sorte que notre base de données soit cohérente avec le modèle. Il suffit de faire un clic droit sur le designer et de choisir de générer la base de données à partir du modèle (voir la figure 37.19).

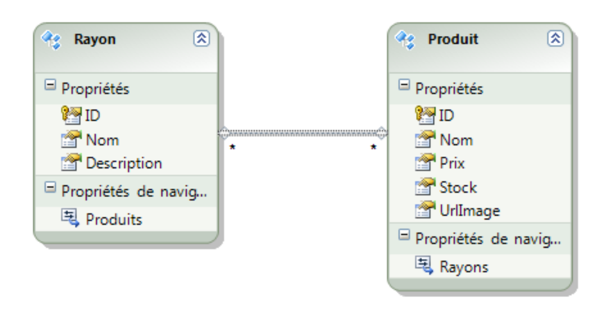


FIGURE 37.17 – La liaison entre les deux entités

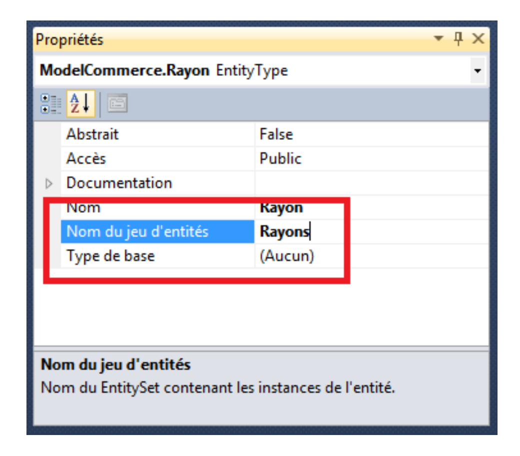


FIGURE 37.18 – Modification du nom du jeu de données

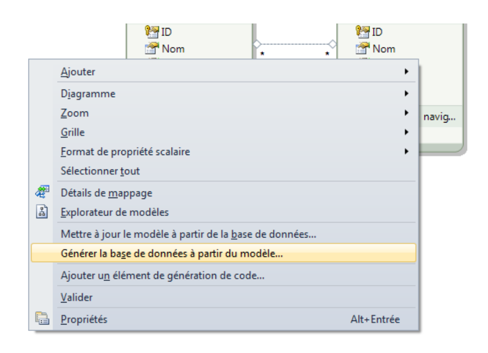


FIGURE 37.19 – Génération de la base de données à partir du modèle

Une nouvelle fenêtre s'ouvre nous permettant de choisir notre source de données. Celle-ci étant vide, cliquez sur `Nouvelle connexion` (voir la figure 37.20).

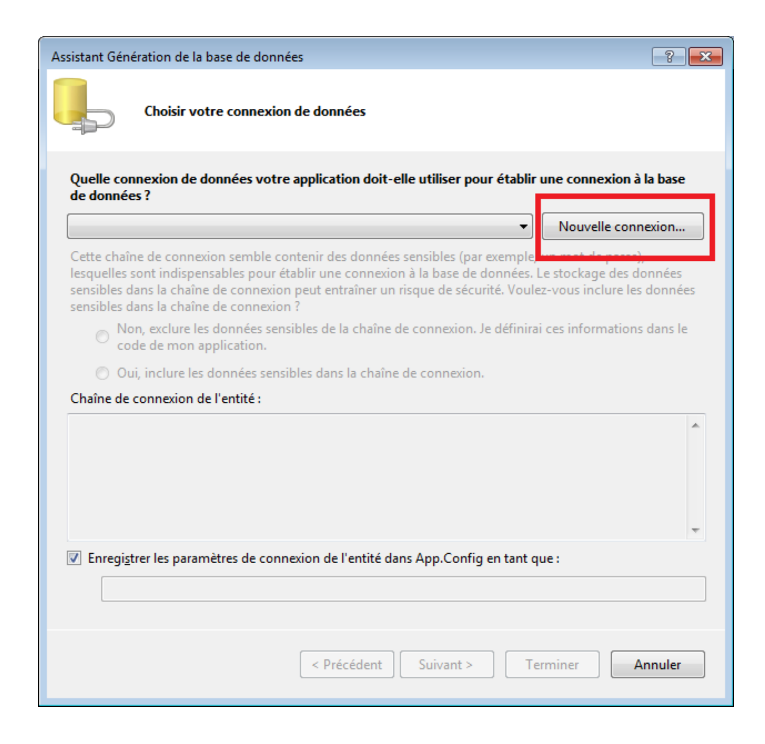


Figure 37.20 – Ajouter une nouvelle connexion

Une nouvelle fenêtre apparaît nous permettant de choisir notre source de données (voir la figure 37.21).

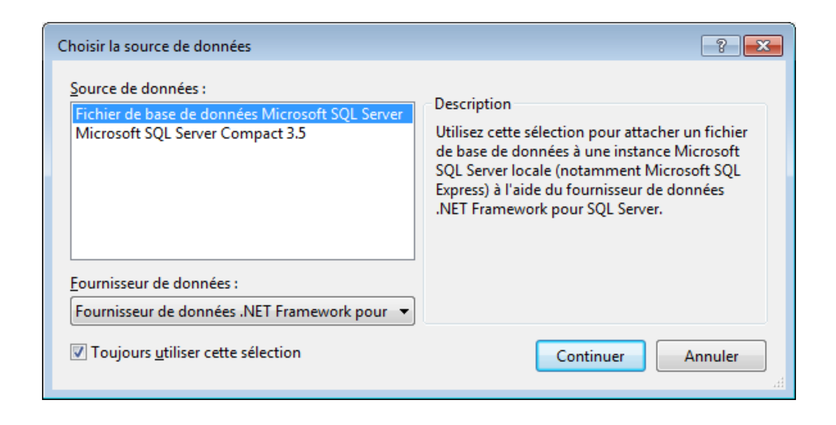


Figure 37.21 – Choix d'un fichier de base de données



Attention, ici dans la version express de Visual C#, il n'est possible de choisir que parmi deux options : un fichier de base de données ou Microsoft SQL Server compact. Dans d'autres versions express (notamment la version permettant de faire du développement web) et dans les versions payantes de Visual Studio, il est possible de choisir directement un serveur de base de données.

Cela aurait été plus pratique. Tant pis, nous allons faire avec ; choisissons le fichier de base de données. Il faut lui donner un emplacement et un nom, par exemple dans le répertoire des projets, je l'appelle `basecommerce.mdf`. Ensuite, pour pouvoir s'y connecter, nous utiliserons l'authentification Windows (voir la figure 37.22).

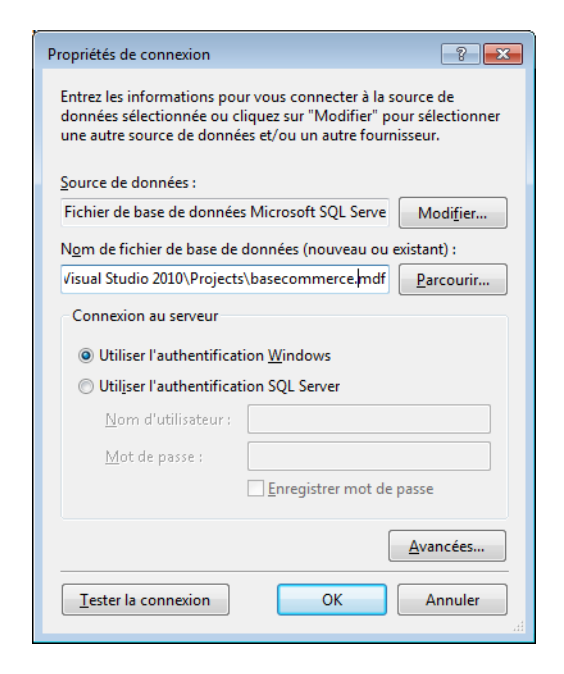


FIGURE 37.22 – Sélection de la base de données et du type de connexion

Au moment de la validation, il nous est demandé si l'on souhaite créer le fichier de base de données. Répondez « oui » !

Puis nous arrivons sur un récapitulatif et nous voyons en bas la chaîne de connexion à la base de données, comme l'illustre la figure 37.23.

Nous pouvons choisir d'enregistrer les paramètres de connexion, ou bien de ne pas le faire, en cochant ou décochant cette case. Dans tous les cas, cette chaîne de connexion ne nous servira pas en l'état. Cliquez sur suivant. Le designer d'Entity Framework nous a finalement créé un fichier contenant des instructions SQL qu'il nous propose d'enregistrer. Ces instructions SQL vont permettre de générer les tables de la base de données (voir la figure 37.24). Nous allons revenir sur ces instructions. Ce fichier s'ouvre également dans Visual C# Express (voir la figure 37.25).

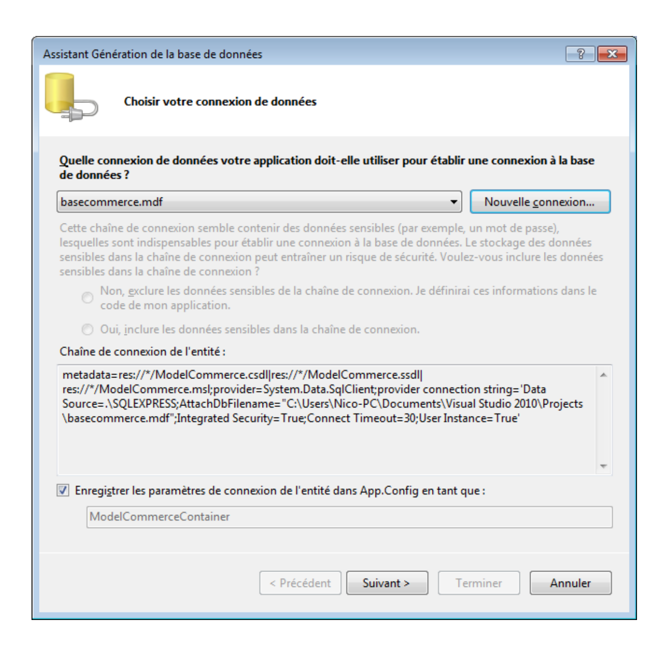


Figure 37.23 – La chaîne de connexion à la base de données générée par Visual C# Express

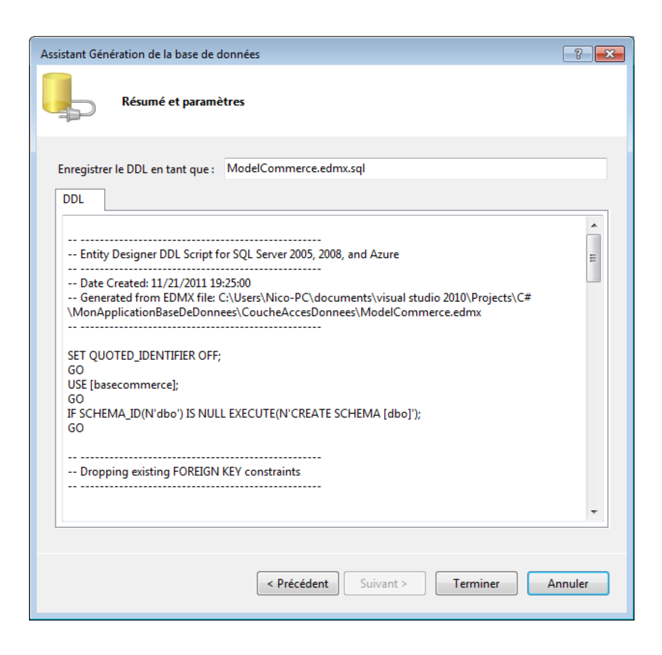


Figure 37.24 – Visual C# Express nous génère un script SQL

FIGURE 37.25 – Le script SQL s'ouvre dans Visual C# Express

Nous en avons terminé pour l'instant avec le designer.

Installer et utiliser l'outil de gestion de BDD

Nous avons le serveur de base de données, qui a été installé en même temps que Visual C# Express. Nous avons le script permettant de générer le modèle de données. Il nous manque un outil permettant de créer la base de données et d'exécuter le script. C'est l'outil de gestion de base de données. Je vous laisse utiliser le code web suivant pour installer « Microsoft® SQL Server® 2008 Management Studio Express » :

▷ Microsoft SQL Server
Code web : 225473

Notez qu'il ne s'agit que des outils puisque nous avons déjà installé un serveur de base de données. Il existe cependant d'autres installations qui cumulent le serveur de base de données ainsi que les outils. Téléchargez l'installeur qui vous convient et exécutez-le. Puis choisissez `Installation` et poursuivez. Choisissez ensuite l'ajout de fonctionnalités à une installation existante, comme indiqué à la figure 37.26.

On poursuit l'installation, puis nous arrivons sur le choix des éléments à installer (voir la figure 37.27).

Même si le nom est trompeur, nous devons effectuer une nouvelle installation de SQL Server 2008. Ce n'est pas tout à fait une installation d'une nouvelle instance car nous avons déjà une instance installée. Nous pouvons le voir sur la copie d'écran, où l'instance existante déjà installée s'appelle `SQLEXPRESS`. Retenons bien le nom de cette instance,

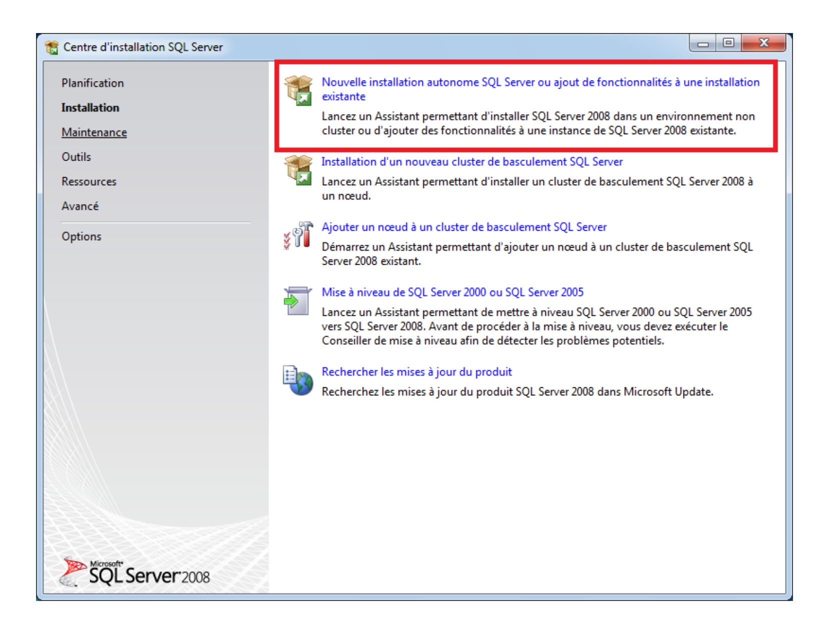


Figure 37.26 – Choix d'une nouvelle installation

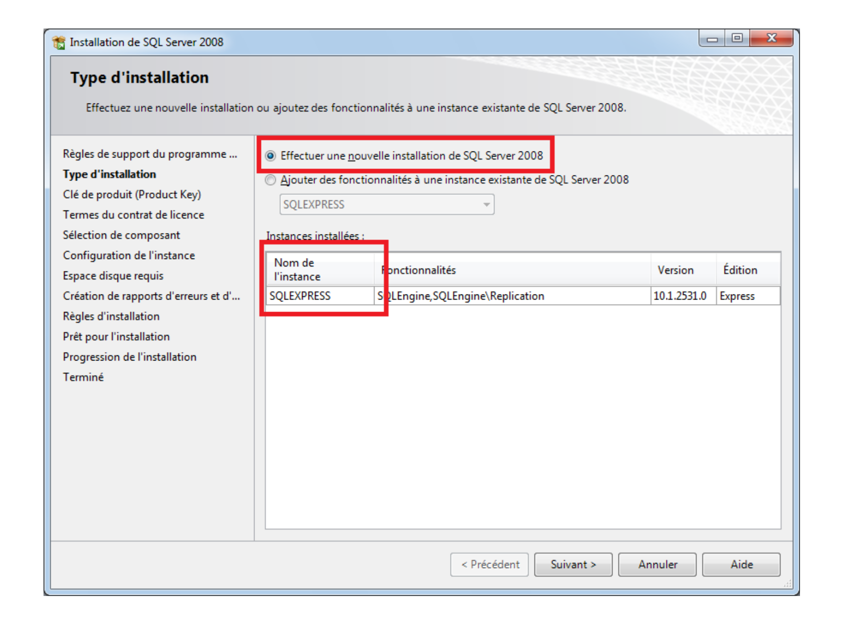


Figure 37.27 – Choix du type d'installation

il nous servira un peu plus loin. Puis nous arrivons sur l'écran suivant qui nous indique que nous allons ajouter l'outil de gestion de base (voir la figure 37.28).

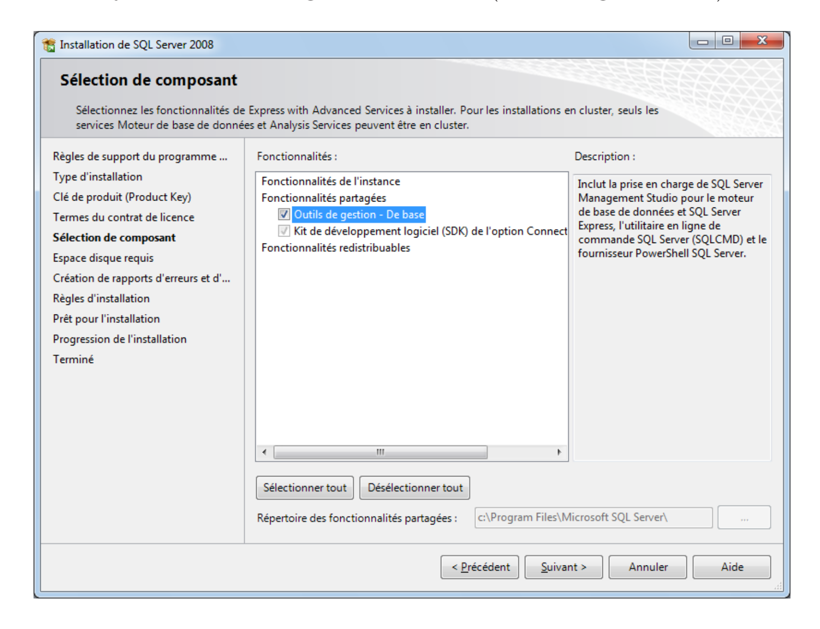


FIGURE 37.28 – Fenêtre de sélection de composants à installer

Et voilà, nous avons terminé cette installation ! Maintenant, nous pouvons enfin démarrer SQL Server Management Studio !

Au démarrage, il nous demande de nous connecter à notre instance de base de données. Par défaut, l'instance s'appelle `SQLEXPRESS`, comme nous l'avons vu, et nous pouvons nous y connecter en la préfixant par le nom de notre machine ou bien simplement en utilisant le point « . », ce qui donne `.\SQLEXPRESS`. Conservez l'authentification Windows (voir la figure 37.29).

Vous arrivez dans l'outil et vous pouvez voir dans l'explorateur d'objets à gauche qu'il n'y a pas (encore) de base de données (voir la figure 37.30).

Nous allons devoir créer notre base de données. Faites un clic droit sur le dossier `Base de données` et choisissez `Nouvelle base de données`. Donnez-lui le même nom que le fichier de base de données que nous avions précédemment créé : `basecommerce`. Validez : la base de données est créée ! Nous la voyons apparaître dans l'explorateur d'objets (voir la figure 37.31) et nous voyons également qu'il n'y a pas de tables dedans.

Cliquons maintenant sur `Nouvelle requête`, une fenêtre vide s'ouvre où nous allons coller le contenu du fichier SQL qui a été généré par le designer de Visual C# Express. Parlons un peu du contenu de ce script. Remarquons déjà que les commentaires sont préfixés par `--`, mais, malins comme nous sommes, nous les aurions reconnus, en plus ils sont en vert ! Ensuite, la ligne suivante permet d'indiquer que nous allons nous positionner sur la base `basecommerce` :

```
USE [basecommerce];
```



FIGURE 37.29 – Fenêtre de connexion à la base de données

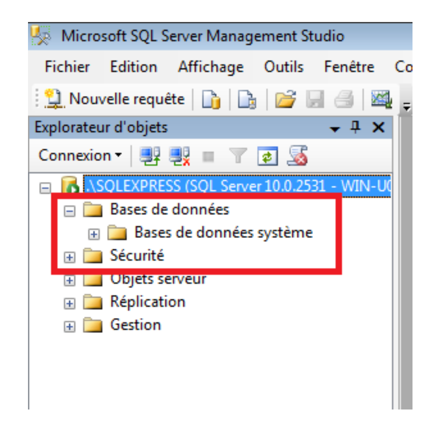


FIGURE 37.30 – Il n'y a aucune base de données

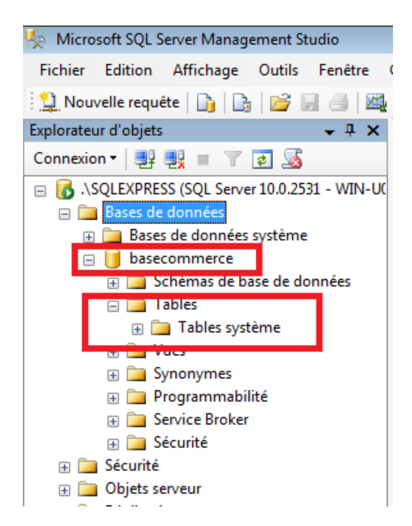


Figure 37.31 – Il n'y a pas de tables dans la base de données que nous avons créée

Si jamais vous n'avez pas donné le même nom à la base de données, c'est ici qu'il faut le changer. Allons un peu plus bas, nous voyons l'instruction :

```
CREATE TABLE [dbo].[Rayons] (
    [ID] int IDENTITY(1,1) NOT NULL,
    [Nom] nvarchar(max)  NOT NULL,
    [Description] nvarchar(max)  NULL
);
```

qui permet de créer la table contenant les rayons, suivi du même genre d'instruction qui permet de créer la table contenant les produits. Sans trop nous attarder dessus, nous pouvons voir la syntaxe permettant de créer la table (avec `CREATE TABLE`) et la syntaxe permettant de créer les champs de la table, ainsi que leurs types.

Après la création des tables, et comme l'indiquent les commentaires pour les anglophones, la suite est une histoire de clé primaire et de clé étrangère.



Clé primaire ? Clé étrangère ? C'est quoi ça ?

Ce sont des notions de base de données. Sans trop entrer dans les détails, je vais vous expliquer rapidement de quoi il s'agit. **Une clé primaire** est une contrainte d'unicité qui permet d'identifier de manière unique un enregistrement dans une table. La clé primaire correspond dans notre cas à l'identifiant d'un rayon ou à l'identifiant d'un produit dans leurs tables respectives. À noter qu'elles ont une propriété complémentaire, à savoir un auto-incrément. C'est-à-dire que c'est SQL Server qui va s'occuper de numéroter automatiquement ces identifiants, en les incrémentant à chaque insertion. **Une clé étrangère** est une contrainte qui garantit l'intégrité référentielle entre

deux tables. Elle identifie une colonne d'une autre table. Cela permet de faire des liens sémantiques entre les tables.



Vous n'avez pas besoin de savoir exactement ce qu'il se passe dans ce script SQL. Nous le regardons vite fait pour la culture, mais il faut juste savoir l'exécuter afin qu'il nous crée les tables.

Ce petit aparté terminé, retournons dans SQL Server Management Studio et collons-y notre requête. Il ne reste plus qu'à exécuter le script en cliquant sur le bouton `Exécuter`, comme indiqué à la figure 37.32.

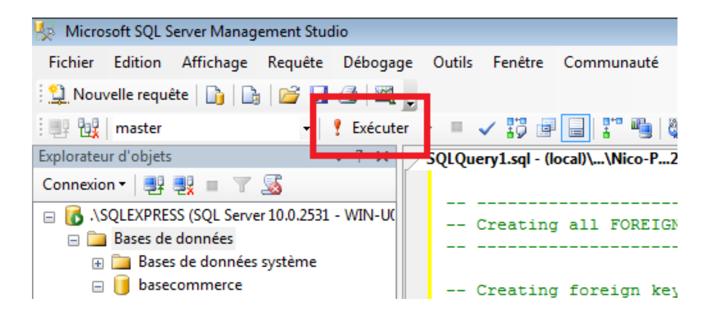


FIGURE 37.32 – Exécuter la requête

Comme tout s'est bien passé, nous pouvons rafraîchir l'explorateur d'objets et constater que les nouvelles tables sont créées (voir la figure 37.33).

Maintenant, nous avons besoin de données dans ces tables. Il y a plusieurs façons de faire. La première est d'utiliser le designer de SQL Server Management Studio, la seconde serait d'utiliser un script SQL, la troisième serait d'utiliser du code C#. Regardons la première solution et faisons un clic droit sur la table `produit` pour « modifier les 200 lignes du haut ». Nous pouvons ensuite insérer des valeurs, comme l'illustre la figure 37.34.

Ne le faites pas, car pour vous éviter du travail, je l'ai fait pour vous grâce à la deuxième méthode, le script SQL :

```
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Télé HD', 299, 50, 'tele.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Console de jeux', 150, 25, 'console.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Canapé', 400, 10, 'canape.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Cuisinière', 280, 20, 'cuisiniere.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Bouilloire', 19, 100, 'bouilloire.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
     VALUES ('Lit 2 places', 149, 15, 'lit.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
```

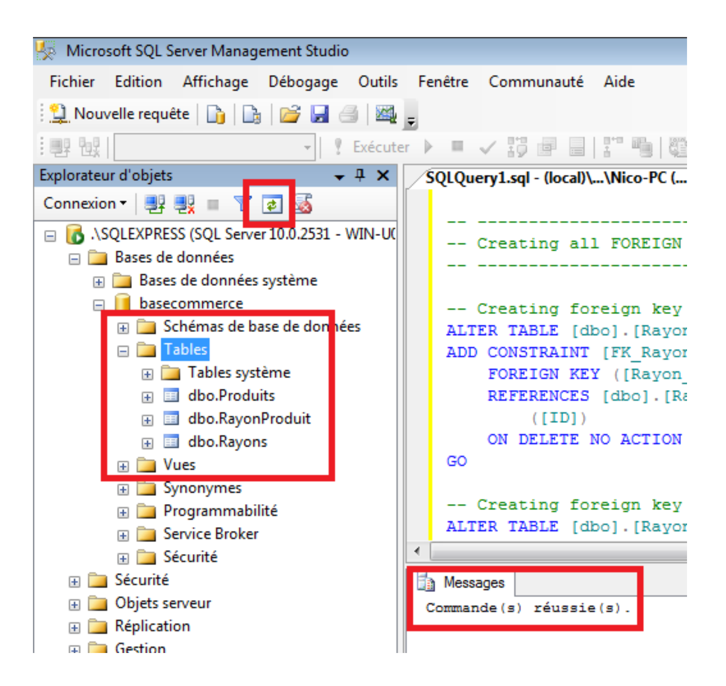


FIGURE 37.33 – Les tables ont correctement été créées

WIN-U0QOL620HAN... - dbo.Produits | SQLQuery1.sql - (local)\...\Nico-PC (...*

	ID	Nom	Prix	Stock	UrlImage
	1	Télé HD	299	50	tele.jpg
	2	Console de jeux	150	25	console.jpg
	3	Canapé	400	10	canape.jpg
	4	Cuisinière	280	20	cuisiniere.jpg
	5	Bouilloire	19	100	bouilloire.jpg
	6	Lit 2 places	149	15	lit.jpg
▸*	NULL	NULL	NULL	NULL	NULL

FIGURE 37.34 – Ajout de données par l'interface de SQL Server Management Studio

```
      VALUES ('Pull', 39.99, 25, 'pull.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
      VALUES ('T-shirt', 19.99, 20, 'tshirt.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
      VALUES ('Pyjama', 15.15, 4, 'pyjama.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
      VALUES ('Tablette PC', 350, 44, 'tablette.jpg')
INSERT INTO [Produits] ([Nom],[Prix],[Stock],[UrlImage])
      VALUES ('Smartphone', 319.99, 40, 'smartphone.jpg')
```

Il vous suffit d'exécuter ce script pour insérer les données. Nous n'allons pas détailler la syntaxe de ce script, mais il est quand même assez facile à lire comme ça. À noter que nous n'avons pas besoin d'indiquer d'identifiant car il est auto-incrémenté par SQL Server. Créons maintenant des rayons, avec la même technique :

```
INSERT INTO [Rayons] ([Nom],[Description])
      VALUES ('Salon', 'Tout ce qu''on trouve dans un salon')
INSERT INTO [Rayons] ([Nom],[Description])
      VALUES ('Cuisine', 'Venez découvrir l''univers de la
          cuisine')
INSERT INTO [Rayons] ([Nom],[Description])
      VALUES ('Dormir', null)
INSERT INTO [Rayons] ([Nom],[Description])
      VALUES ('Hi-Tech', 'Les produits hi-tech ...')
INSERT INTO [Rayons] ([Nom],[Description])
      VALUES ('Vêtements', null)
```

Vous pouvez copier ces codes grâce au code web suivant :

▷ Copier ces codes
Code web : 174353

Nous pouvons voir le contenu de ces tables en faisant un clic droit, puis `Sélectionner les 1000 lignes du haut`, comme illustré à la figure 37.35.

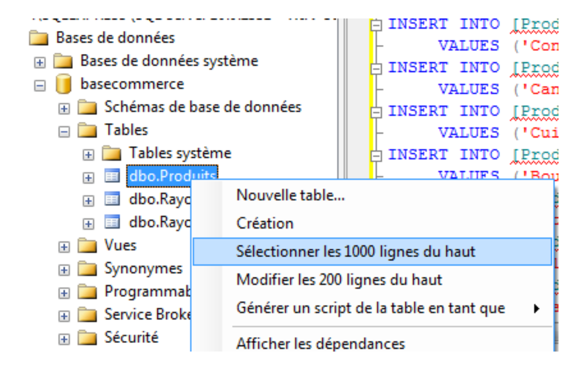


FIGURE 37.35 – Afficher les premières lignes de la table

Ce qui permet de voir la table `Produits` (voir la figure 37.36).

Résultats | Messages

	ID	Nom	Prix	Stock	UrlImage
1	1	Télé HD	299	50	tele.jpg
2	2	Console de jeux	150	25	console.jpg
3	3	Canapé	400	10	canape.jpg
4	4	Cuisinière	280	20	cuisiniere.jpg
5	5	Bouilloire	19	100	bouilloire.jpg
6	6	Lit 2 places	149	15	lit.jpg
7	7	Pull	40	25	pull.jpg
8	8	T-shirt	20	20	tshirt.jpg
9	9	Pyjama	15	4	pyjama.jpg
10	10	Tablette PC	350	44	tablette.jpg
11	11	Smartphone	320	40	smartphone.jpg

FIGURE 37.36 – Les premières lignes de la table `Produits`

Il n'y a plus qu'à relier les produits et les rayons. Pour cela, il faut relier les identifiants entre eux. Par exemple, avec le script suivant j'indique que le rayon `Salon` (identifiant 1) contient la télé HD (identifiant 1), la console de jeux (identifiant 2), le canapé (identifiant 3), la tablette PC (identifiant 10). J'indique également que le rayon `Cuisine` (identifiant 2) contient la cuisinière (identifiant 4), ainsi que la bouilloire (identifiant 5). Et ainsi de suite. . . :

```
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (1, 1)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (1, 2)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (1, 3)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (1, 10)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (2, 4)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (2, 5)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (3, 3)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (3, 6)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (3, 9)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (4, 1)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (4, 2)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (4, 10)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (4, 11)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
```

```
     VALUES (5, 7)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (5, 8)
INSERT INTO [RayonProduit] ([Rayons_ID],[Produits_ID])
     VALUES (5, 9)
```

Vous pouvez copier ce code grâce au code web suivant :

▷ Copier ce code
Code web : 921140

Voilà, notre base de données est prête. Nous allons pouvoir utiliser cette base et ses données dans notre code C# !

Se connecter à la base de données, lire et écrire

Il est temps de se connecter à notre base de données depuis notre application C#. Pour cela, nous avons besoin de la chaîne de connexion à la base de données. Nous avons déjà parlé de la chaîne de connexion, elle contient toutes les informations nécessaires pour se connecter à la base de données ; à savoir le nom du serveur, le nom de la base, les identifiants de connexion, le type de connexion, etc. Nous en avons également vu un aperçu lorsque nous avons utilisé l'assistant de génération de modèle sauf que je vous ai indiqué que cette chaîne de connexion n'allait pas être bonne pour nos besoins. En effet, nous avons besoin qu'elle pointe vers notre serveur de base de données et pas vers le fichier temporaire que nous avons créé pour les besoins de l'assistant. Cette chaîne de connexion a toute sa place dans le fichier de configuration de l'application, que nous avons déjà étudié. Si vous ne l'avez pas déjà ajouté, il est temps de le faire. Et vous pouvez le remplir avec la configuration suivante :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="NotreBaseDeDonnees" connectionString="metadata=
        res://*/ModelCommerce.csdl|res://*/ModelCommerce.ssdl|
        res://*/ModelCommerce.msl;provider=System.Data.SqlClient
        ;provider connection string='data source=.\SQLEXPRESS;
        Initial Catalog=basecommerce;integrated security=True'"
        providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Nous indiquons ici que notre chaîne de connexion va être accessible par le nom `NotreBaseDeDonnees`. Il y a plein d'informations dans l'attribut `connectionString`, mais ce qui nous intéresse surtout, c'est d'indiquer la source de données (à savoir : `data source=.\SQLEXPRESS`) ce qui va nous permettre d'indiquer que notre serveur est accessible à cette adresse, puis le nom de la base que nous avons créée (`Initial Catalog=basecommerce`) et enfin d'indiquer que nous utilisons l'authentification Windows (`integrated security=True`). Le reste permet de donner des informations de

description du modèle. Enfin, nous indiquons que nous utilisons les méthodes d'Entity Framework pour l'accès aux données, à travers `System.Data.EntityClient`. Bref, beaucoup de ces informations sont issues de la chaîne de connexion générée par Visual C# Express ; nous avons simplement changé le mode de connexion pour qu'il corresponde à nos besoins. Maintenant, nous allons pouvoir accéder à la chaîne de connexion avec le `ConfigurationManager` que nous connaissons bien désormais :

```
string chaineConnexion = ConfigurationManager.ConnectionStrings
    ["NotreBaseDeDonnees"].ConnectionString;
```

Vous n'avez bien sûr pas oublié de référencer l'assembly `System.Configuration` ! Retournons dans notre designer et cliquons dessus pour observer les propriétés du modèle. Nous allons modifier le `nom du conteneur d'entités` pour y mettre un nom un peu plus parlant, à savoir `BaseDeDonnees` (voir la figure 37.37).

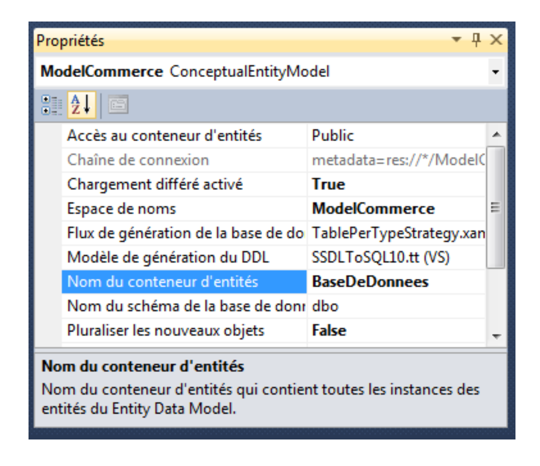


FIGURE 37.37 – Modification du nom du conteneur d'entités

C'est le point d'entrée de notre accès aux données. Il s'agit en fait d'une classe qui a été générée par le designer d'Entity Framework.



Une classe générée ? Où ça ?

Dans l'explorateur de solutions, en dépliant le fichier `ModelCommerce.edmx`. Il s'agit d'un fichier intitulé `ModelCommerce.Designer.cs` qui contient la définition des classes générées et toute la logique permettant d'accéder à la base de données. Nous pouvons l'ouvrir, mais le code est assez verbeux et nous risquons de nous perdre. Faisons confiance à Visual C# Express, nous allons utiliser son code généré les yeux fermés ! Notez quand même que le code généré possède des classes préfixées par le mot-clé `partial`. Je vais y revenir plus loin.

Vous ne vous en rendez peut-être pas encore compte, mais l'outil d'ORM « Entity Framework » nous simplifie énormément la tâche (et je pèse mes mots, **énormément** !),

car non seulement il génère toutes les classes représentant les données en base (comme la classe `Produit` ou la classe `Rayon`) mais il s'occupe également de nous simplifier la création, la lecture ou la modification des données en base. Si nous avions dû le faire à la main comme c'était le cas avant l'utilisation d'ORMs, cela aurait mérité une centaine de pages supplémentaires d'explications et de code à comprendre. Là, nous nous positionnons en tant qu'utilisateur de ces classes générées et vous allez voir que c'est facile à utiliser ; vous n'imaginez pas le plaisir que c'est de constater que l'ORM a travaillé pour nous !

Nous pouvons désormais instancier la classe `BaseDeDonnees` générée en lui passant en paramètre la chaîne de connexion :

```
string chaineConnexion = ConfigurationManager.ConnectionStrings
    ["NotreBaseDeDonnees"].ConnectionString;
BaseDeDonnees baseDeDonnees = new BaseDeDonnees(chaineConnexion
    );
```

Vous voilà connectés à la base de données. Nous allons pouvoir utiliser les objets que Visual C# Express a générés à travers cette variable de type `BaseDeDonnees`, comme par exemple la propriété `Rayons` qui nous permet d'accéder aux rayons de notre base :

```
foreach (Rayon rayon in baseDeDonnees.Rayons)
{
    Console.WriteLine("{0} ({1})", rayon.Nom, rayon.Description
        );
}
```

Ici, nous pouvons parcourir la liste des rayons avec un `foreach` car la propriété `Rayons` est du type `ObjectSet<>` qui implémente `IEnumerable<>`.

Ce qui donne :

```
Salon (Tout ce qu'on trouve dans un salon)
Cuisine (Venez découvrir l'univers de la cuisine)
Dormir ()
Hi-Tech (Les produits hi-tech ...)
Vêtements ()
```

De même, nous pouvons parcourir tous les produits grâce à la propriété `Produits` :

```
foreach (Produit produit in baseDeDonnees.Produits)
{
    Console.WriteLine("{0} : {1}", produit.Nom, produit.Prix);
}
```

Ce qui donne :

```
Télé HD : 299
Console de jeux : 150
Canapé : 400
Cuisinière : 280
Bouilloire : 19
```

```
Lit 2 places : 149
Pull : 40
T-shirt : 20
Pyjama : 15
Tablette PC : 350
Smartphone : 320
```

Et ce qui est formidable, c'est qu'étant donné que la propriété `Produits` est énumérable, nous allons pouvoir y faire toutes les requêtes LINQ que nous le souhaitons, par exemple :

```
IEnumerable<Produit> produits = from produit in baseDeDonnees.Produits
                     where produit.Prix > 150
                     orderby produit.Prix descending
                     select produit;
foreach (Produit produit in produits)
{
    Console.WriteLine("{0} : {1}", produit.Nom, produit.Prix);
}
```

Nous obtenons tous les produits dont le prix est supérieur à 150, triés par prix décroissant :

```
Canapé : 400
Tablette PC : 350
Smartphone : 320
Télé HD : 299
Cuisinière : 280
```

Pratique ! Tout le SQL nécessaire pour renvoyer cette liste de produits filtrée a été généré par Entity Framework. Nous n'avons rien à faire d'autre que d'utiliser le C#.

Et voilà. Avouez que c'est quand même super simple, non ? Avouez également que, si vous avez l'habitude de tout faire à la main dans un autre langage de programmation, vous êtes émerveillés ! J'exagère peut-être un peu, mais Entity Framework nous fait gagner un temps considérable au développement ainsi que tout au long de la vie de l'application.

Merci à lui de nous avoir généré tout le code adéquat. À propos de génération de code, souvenez-vous que les classes générées sont partielles. Nous en avons déjà parlé dans le chapitre dédié, mais je vous rappelle le but ici. Il s'agit de permettre d'ajouter des fonctionnalités à la classe sans avoir à modifier le fichier `ModelCommerce.Designer.cs`. En effet, à chaque fois que nous faisons une modification sur notre modèle (ajout d'entité, changement de nom, etc.), il régénère toutes les classes de ce fichier. Si nous avions modifié des choses à la main dedans, elles vont disparaître... Le mot-clé `partial` nous offre l'opportunité d'ajouter des fonctionnalités à la classe depuis un autre fichier. Nous pouvons en profiter pour rajouter nos propres méthodes, par exemple une méthode qui renvoie les produits dont le prix est supérieur à un prix passé en paramètre. Il suffit de

déclarer une classe partielle du même nom que la classe `BaseDeDonnees`, située dans le même espace de noms et de rajouter des méthodes. Par exemple :

```
public partial class BaseDeDonnees
{
    public IEnumerable<Produit> ProduitsPlusCherQue(decimal prix)
    {
        return from produit in Produits
               where produit.Prix > prix
               select produit;
    }
}
```

Nous pourrons donc utiliser cette méthode de cette façon :

```
foreach (Produit produit in baseDeDonnees.ProduitsPlusCherQue(200))
{
    Console.WriteLine("{0} : {1}", produit.Nom, produit.Prix);
}
```

Ce qui donnera :

```
Télé HD : 299
Canapé : 400
Cuisinière : 280
Tablette PC : 350
Smartphone : 320
```

Remarquons que chaque `Rayon` possède également une propriété `Produits` ; c'est la propriété de navigation que nous avons renommée précédemment. Entity Framework a donc compris qu'il y avait une relation entre les rayons et les produits et il permet d'accéder aux produits qui font partie du rayon, grâce à cette propriété. Ainsi, nous pouvons écrire un code, comme le suivant, qui accède à la propriété `Produits` d'un rayon et permet d'afficher la liste de tous les produits de chaque rayon :

```
foreach (Rayon rayon in baseDeDonnees.Rayons)
{
    Console.WriteLine("{0} ({1})", rayon.Nom, rayon.Description);
    foreach (Produit produit in rayon.Produits)
    {
        Console.WriteLine("\t{0} : {1}", produit.Nom, produit.Prix);
    }
}
```

Sauf qu'ici nous rencontrons un problème. Si nous exécutons ce bout de code, nous aurons l'exception suivante :

```
Exception non gérée : System.Data.
    EntityCommandExecutionException: Une erreur s'est produite
    lors de l'exécution de la définition de la commande. Pour
    plus de détails, consultez l'exception interne. ---> System.
    InvalidOperationException: Un DataReader associé à cette
    Command est déjà ouvert. Il doit d'abord être fermé.
[...]
```



Pourquoi un tel problème ?

En fait, cela vient de la façon dont sont récupérées les données. Lorsque nous accédons à la propriété `Rayons`, Entity Framework génère une requête en base de données pour récupérer la liste des rayons. Puis à l'intérieur de la boucle, lorsque nous accédons à la propriété `Produits` d'un rayon, il génère à nouveau une requête pour récupérer les produits de ce rayon. Il y a donc deux connexions à la base de données en même temps, et ça, il ne sait pas le faire par défaut.

Il y a plusieurs façons de corriger le problème. La première est de changer la chaîne de connexion en rajoutant une directive permettant de préciser qu'on autorise l'accès multiple, à savoir :

```
multipleactiveresultsets=True;
```

La chaîne de connexion devient donc :

```
<add name="NotreBaseDeDonnees" connectionString="metadata=res
    ://*/ModelCommerce.csdl|res://*/ModelCommerce.ssdl|res://*/
    ModelCommerce.msl;provider=System.Data.SqlClient;provider
    connection string='data source=.\SQLEXPRESS;Initial Catalog=
    basecommerce;integrated security=True;
    multipleactiveresultsets=True;'" providerName="System.Data.
    EntityClient" />
```

Avec ce changement, si nous exécutons ce code, nous aurons :

```
Salon (Tout ce qu'on trouve dans un salon)
        Télé HD : 299
        Console de jeux : 150
        Canapé : 400
        Tablette PC : 350
Cuisine (Venez découvrir l'univers de la cuisine)
        Cuisinière : 280
        Bouilloire : 19
Dormir ()
        Canapé : 400
        Lit 2 places : 149
        Pyjama : 15
```

```
Hi-Tech (Les produits hi-tech ...)
        Télé HD : 299
        Console de jeux : 150
        Tablette PC : 350
        Smartphone : 320
Vêtements ()
        Pull : 40
        T-shirt : 20
        Pyjama : 15
```

La deuxième solution est de faire en sorte que la première requête soit terminée avant l'exécution des suivantes. Pour cela, il suffit de forcer l'évaluation de la requête en utilisant par exemple un `ToList()` :

```
foreach (Rayon rayon in baseDeDonnees.Rayons.ToList())
{
    Console.WriteLine("{0} ({1})", rayon.Nom, rayon.Description);
    foreach (Produit produit in rayon.Produits)
    {
        Console.WriteLine("\t{0} : {1}", produit.Nom, produit.Prix);
    }
}
```

Ceci est possible, car Entity Framework bénéficie de l'exécution différée ; le `ToList()` résout le problème en forçant l'exécution de la requête.

Enfin, la dernière solution est de faire en sorte que la première requête qui charge les `Rayons` inclue également le chargement des produits. Ainsi, il n'y a qu'une seule et unique requête qui charge tout. Cela se passe avec la méthode `Include`, en précisant le nom de la propriété de navigation à charger :

```
foreach (Rayon rayon in baseDeDonnees.Rayons.Include("Produits"))
{
    Console.WriteLine("{0} ({1})", rayon.Nom, rayon.Description);
    foreach (Produit produit in rayon.Produits)
    {
        Console.WriteLine("\t{0} : {1}", produit.Nom, produit.Prix);
    }
}
```

Il y a plusieurs choses à remarquer ici.

Nous avons vu qu'Entity Framework est capable d'aller chercher les données liées entre elles grâce aux propriétés de navigation. Les deux premiers scénarios couverts tirent parti de ce qu'on appelle le *lazy loading* que l'on peut traduire en « chargement paresseux ». Cela veut dire que c'est uniquement lorsque l'on accède à la propriété `Produits`

qu'Entity Framework va aller lire le contenu associé en base de données. Ceci implique qu'à chaque tentative d'accès à la propriété `Produits` d'un rayon, Entity Framework va effectuer une requête en base de données pour ramener les produits concernés. C'est très bien si on fait ça une ou deux fois, mais dans notre cas au final on fait autant de requête qu'il y a de rayons. Ce qui n'est pas très performant... Le troisième scénario montre l'utilisation de la méthode `Include` qui permet de tout rapatrier en une seule requête, ce qui est évidemment plus performant.



Faites bien attention à votre utilisation d'Entity Framework. Si ce n'est pas très grave pour une petite application, cela peut le devenir pour des applications avec des grosses bases de données.

Alors, vous ne trouvez pas que la lecture en base de données est particulièrement aisée ? Merci Entity Framework ! Notons que nous pouvons également accéder aux rayons dans lesquels sont positionnés les produits grâce à la propriété `Rayons`. Nous pourrions éventuellement nous en servir pour afficher le nombre de rayons dans lesquels le produit est présent.

L'écriture en base de données est tout aussi aisée. Le principe est d'ajouter des valeurs à notre objet de base de données et de sauvegarder les modifications.

Pour ajouter un nouveau rayon, il suffit d'appeler la méthode `AddObject` disponible sur la propriété `Rayons`. Il ne faudra pas oublier d'appeler la méthode `SaveChanges` qui s'occupe d'insérer physiquement les valeurs en base de données. Pour créer un nouveau rayon, il suffira d'instancier un objet `Rayon`, de renseigner des propriétés, de créer des produits et de les ajouter au rayon, par exemple :

```
Rayon rayon = new Rayon();
rayon.Nom = "Vins";
rayon.Description = "Venez découvrir notre sélection des plus
    grands châteaux";

Produit produit1 = new Produit();
produit1.Nom = "Château ronto";
produit1.Prix = 9.99M;
produit1.Stock = 60;
produit1.UrlImage = "à compléter ...";

Produit produit2 = new Produit();
produit2.Nom = "Château toro";
produit2.Prix = 15;
produit2.Stock = 6;
produit2.UrlImage = "à compléter ...";

rayon.Produits.Add(produit1);
rayon.Produits.Add(produit2);

baseDeDonnees.Rayons.AddObject(rayon);
baseDeDonnees.SaveChanges();
```

Ainsi, si nous réaffichons la liste des rayons, nous pourrons voir un rayon de plus contenant des produits en plus. . .

```
Salon (Tout ce qu'on trouve dans un salon)
        Télé HD : 299
        Console de jeux : 150
        Canapé : 400
        Tablette PC : 350
Cuisine (Venez découvrir l'univers de la cuisine)
        Cuisinière : 280
        Bouilloire : 19
Dormir ()
        Canapé : 400
        Lit 2 places : 149
        Pyjama : 15
Hi-Tech (Les produits hi-tech ...)
        Télé HD : 299
        Console de jeux : 150
        Tablette PC : 350
        Smartphone : 320
Vêtements ()
        Pull : 40
        T-shirt : 20
        Pyjama : 15
Vins (Venez découvrir notre sélection des plus grands châteaux)
        Château ronto : 10
        Château toro : 15
```

De même, si vous allez voir en base de données, vous aurez bien les nouveaux éléments, ainsi que l'illustre la figure 37.38.

On observe la création d'un nouveau rayon, de deux nouveaux produits et nous avons bien dans la table de relation les nouveaux produits reliés au nouveau rayon. Il est également possible d'ajouter un produit à un rayon. Nous pouvons le faire de deux manières différentes. La première est d'ajouter un `Produit` directement dans la collection `Produits` d'un rayon, Il sera directement ajouté dans le rayon de notre choix. La deuxième est d'ajouter un produit dans la collection `Produits` et si nous voulons qu'il soit présent dans un rayon, il faudra que sa propriété `Rayons` contienne les rayons dans lesquels nous souhaitons ajouter le produit. Voyons la première méthode :

```
Rayon rayon = baseDeDonnees.Rayons.First(r => r.Nom == "Vins");

Produit produit = new Produit();
produit.Nom = "Chateau pinière";
produit.Prix = 12.50M;
produit.Stock = 40;
produit.UrlImage = "vin.jpg";
produit.Rayons.Add(rayon);

baseDeDonnees.SaveChanges();
```

Résultats | Messages

1	1	Salon	Tout ce qu'on trouve dans un salon
2	2	Cuisine	Venez découvrir l'univers de la cuisine
3	3	Dormir	NULL
4	4	Hi-Tech	Les produits hi-tech ...
5	5	Vêtements	NULL
6	6	Vins	Venez découvrir notre sélection des plus grands ...

	ID	Nom	Prix	Stock	UrlImage
8	8	T-shirt	20	20	tshirt.jpg
9	9	Pyjama	15	4	pyjama.jpg
10	10	Tablette PC	350	44	tablette.jpg
11	11	Smartphone	320	40	smartpho...
12	12	Château ronto	10	60	à complét...
13	13	Château toro	15	6	à complét...

	Rayons_ID	Produits_ID
11	4	2
12	4	10
13	4	11
14	5	7
15	5	8
16	5	9
17	6	12
18	6	13

FIGURE 37.38 – Le nouveau rayon est les nouveaux produits sont visibles dans les tables

Nous commençons par récupérer un rayon, puis nous instancions un objet de type `Produit`. Enfin, nous faisons le lien entre le produit et le rayon en ajoutant le rayon à la collection `Rayons` de notre objet produit. Comme d'habitude, la méthode `SaveChanges()` permet de faire persister les informations.

La seconde méthode est un peu plus simple à appréhender ; il suffit d'instancier un objet `Produit` et de l'ajouter à la collection `Produits` d'un rayon :

```
Rayon rayon = baseDeDonnees.Rayons.First(r => r.Nom == "Vins");

Produit produit = new Produit();
produit.Nom = "Chateau réro";
produit.Prix = 3.20M;
produit.Stock = 10;
produit.UrlImage = "vin1.jpg";

rayon.Produits.Add(produit);

baseDeDonnees.SaveChanges();
```

Dans les deux cas, Entity Framework arrive à faire le lien entre un rayon et un produit. Le château pinière et le château réro ont bien été ajoutés. . . ! Vous pouvez également vérifier en base de données que la relation entre le rayon et le produit a bien été faite.

Il est également possible de modifier des enregistrements en base de données. Le principe est de modifier l'élément concerné dans l'objet de base de données et d'appeler la

méthode `SaveChanges()` :

```
Rayon rayon = baseDeDonnees.Rayons.First(r => r.Nom == "Vins");
rayon.Description = "Les meilleurs vins";
baseDeDonnees.SaveChanges();
```

Enfin, nous pouvons aussi supprimer des données en base. Le principe est le même que pour l'ajout. Nous appelons une méthode qui s'occupe de la suppression et nous appelons la méthode `SaveChanges`. Par contre, il faut faire attention à l'intégrité des données. On ne peut pas supprimer un rayon qui a des produits dedans. Il faut commencer par retirer la relation entre les produits et le rayon :

```
Rayon rayon = baseDeDonnees.Rayons.First(r => r.Nom == "Vins");
rayon.Produits.Clear();
baseDeDonnees.DeleteObject(rayon);
baseDeDonnees.SaveChanges();
```

Le fait d'appeler la méthode `Clear()` sur les produits du rayon vide le rayon de ses produits. Il n'y a donc plus de produits dans ce rayon, mais ils existent toujours en base de données car ils n'ont pas été supprimés physiquement. C'est important car ces produits peuvent également être présents dans d'autres rayons, nous ne pouvons donc pas les supprimer. Ensuite, on utilise la méthode `DeleteObject` pour supprimer un élément de la base de données. Ici, nous supprimons le rayon et nous validons les modifications avec `SaveChanges()`.

Notez que si nous n'avions pas vidé le rayon de ses produits, la suppression du rayon aurait été impossible car comme il existe une relation entre les produits et les rayons et que notre base de données possède une contrainte d'intégrité (la clé étrangère) entre les produits et le rayon, la suppression aurait provoqué une erreur. En effet, la table `RayonProduit` contiendrait un identifiant de rayon qui n'existe plus. C'est impossible ! La contrainte de la clé étrangère est là pour nous assurer que nous ne faisons pas n'importe quoi dans la base de données et qu'elle est toujours cohérente. Si nous l'avions fait, nous aurions eu l'exception suivante :

```
Exception non gérée : System.Data.UpdateException: Une erreur s'
    est produite lors de la mise à jour des entrées. Pour plus d'
    informations, consultez l'exception interne. ---> System.Data
    .SqlClient.SqlException: L'instruction DELETE est en conflit
    avec la contrainte REFERENCE "FK\_RayonProduit\_Rayon". Le
    conflit s'est produit dans la base de données "basecommerce",
     table "dbo.RayonProduit", column 'Rayons\_ID'.
L'instruction a été arrêtée. [...]
```

Remarquez que nous aurons désormais des produits orphelins. Tout dépend de ce que l'on veut faire maintenant. Souhaitons-nous qu'ils soient supprimés également vu qu'ils ne sont plus dans aucun rayon ? Souhaitons-nous qu'ils restent présents pour pouvoir les ajouter ultérieurement à un autre rayon ? Ça, c'est vous qui décidez ! Maintenant que vous savez supprimer des objets, vous pouvez faire comme bon vous semble.

Notons avant de terminer qu'il est tout à fait possible de faire plusieurs ajouts, modifications ou suppressions en même temps. Il suffira de terminer toutes les instructions

par la méthode `SaveChanges()` qui s'arrangera pour tout faire persister !

En résumé

- Entity Framework est l'outil de mapping objet relationnel de Microsoft permettant de travailler sur une base de données relationnelle avec une approche orientée objet.
- Entity Framework est capable de modéliser des données et de générer les tables correspondantes en base de données sans qu'il soit nécessaire de maîtriser le SQL.
- Il simplifie grandement la lecture et l'écriture des données en base et tire parti, si besoin, des mécanismes de chargement paresseux.

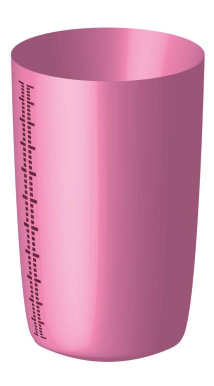
Chapitre 38

Les tests unitaires

Difficulté : 

Une des grandes préoccupations des créateurs de logiciels est d'être certains que leur application informatique fonctionne et surtout qu'elle fonctionne dans toutes les situations possibles. Nous avons tous déjà vu notre système d'exploitation planter, ou bien notre logiciel de traitement de texte nous faire perdre les 50 pages de rapport que nous étions en train de taper. Ou encore, un élément inattendu dans un jeu où l'on arrive à passer à travers un mur alors qu'on ne devrait pas...

Bref, pour être sûr que son application fonctionne, il faut faire des tests.



Qu'est-ce qu'un test unitaire et pourquoi en faire ?

Un test constitue une façon de vérifier qu'un système informatique fonctionne.

Tester son application c'est bien. Il faut absolument le faire. C'est en général une pratique plutôt laissée de côté, car rébarbative. Il y a plusieurs façons de faire des tests. Celle qui semble la plus naturelle est celle qui se fait manuellement. On lance son application, on clique partout, on regarde si elle fonctionne. Celle que je vais présenter ici constitue une pratique automatisée visant à s'assurer que des bouts de code fonctionnent comme il faut et que tous les scénarios d'un développement sont couverts par un test. Lorsque les tests couvrent tous les scénarios d'un code, nous pouvons assurer que notre code fonctionne. De plus, cela permet de faire des opérations de maintenance sur le code tout en étant certain qu'il n'aura pas subi de régressions. De la même façon, les tests sont un filet de sécurité lorsqu'on souhaite refactoriser son code ou l'optimiser. Cela permet dans certains cas d'avoir un guide pendant le développement, notamment lorsqu'on pratique le TDD. Le *Test Driven Development* (TDD) [1] est une méthode de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel. Nous y reviendrons ultérieurement.



Un test est donc un bout de code qui permet de tester un autre code.

En général, un test se décompose en trois parties, suivant le schéma « AAA », qui correspond aux mots anglais *Arrange, Act, Assert*, que l'on peut traduire en français par : Arranger, Agir, Auditer.

- Arranger : il s'agit dans un premier temps de définir les objets, les variables nécessaires au bon fonctionnement de son test (initialiser les variables, initialiser les objets à passer en paramètres de la méthode à tester, etc.).
- Agir : ensuite, il s'agit d'exécuter l'action que l'on souhaite tester (en général, exécuter la méthode que l'on veut tester, etc.).
- Auditer : enfin, il faut vérifier que le résultat obtenu est conforme à nos attentes.

Notre premier test

Imaginons que nous voulions tester une méthode toute simple qui fait l'addition entre deux nombres, par exemple la méthode suivante :

```
public static int Addition(int a, int b)
{
    return a + b;
}
```

Faire un test consiste à écrire des bouts de code permettant de s'assurer que le code fonctionne. Cela peut-être par exemple :

1. En français, le TDD se dit « développement piloté par les tests ».

```
static void Main(string[] args)
{
    // arranger
    int a = 1;
    int b = 2;
    // agir
    int resultat = Addition(a, b);
    // auditer
    if (resultat != 3)
        Console.WriteLine("Le test a raté");
}
```

Ici, le test passe bien, ouf ! Pour être complet, le test doit couvrir un maximum de situations ; il faut donc tester notre code avec d'autres valeurs, et ne pas oublier les valeurs limites :

```
static void Main(string[] args)
{
    int a = 1;
    int b = 2;
    int resultat = Addition(a, b);
    if (resultat != 3)
        Console.WriteLine("Le test a raté");
    a = 0;
    b = 0;
    resultat = Addition(a, b);
    if (resultat != 0)
        Console.WriteLine("Le test a raté");
    a = -5;
    b = 5;
    resultat = Addition(a, b);
    if (resultat != 0)
        Console.WriteLine("Le test a raté");
}
```

Voilà pour le principe. Ici, nous considérons avoir écrit suffisamment de tests pour nous assurer que cette méthode est bien fonctionnelle. Bien sûr, cette méthode était par définition fonctionnelle, mais il est important de prendre le réflexe de tester des fonctionnalités qui sont déterminantes pour notre application.

Voyons maintenant comment nous pourrions tester une méthode avec l'approche TDD. Pour rappel, lors d'une approche TDD, le but est de pouvoir faire un développement à partir des cas de tests préalablement établis par la personne qui exprime le besoin ou suivant les spécifications fonctionnelles.

Imaginons que nous voulions tester une méthode qui calcule la factorielle d'un nombre. Nous savons que la factorielle de 0 vaut 1, la factorielle de 1 vaut 1. Commençons par écrire les tests :

```
static void Main(string[] args)
{
    int valeur = 0;
```

```
    int resultat = Factorielle(valeur);
    if (resultat != 1)
        Console.WriteLine("Le test a raté");

    valeur = 1;
    resultat = Factorielle(valeur);
    if (resultat != 1)
        Console.WriteLine("Le test a raté");
}
```

Le code ne compile pas ! Forcément, nous n'avons pas encore créé la méthode `Factorielle`. C'est la première étape. La suite de la méthode est de faire en sorte que le test compile, mais il échouera puisque la méthode n'est pas encore implémentée :

```
public static int Factorielle(int a)
{
    throw new NotImplementedException();
}
```

Il faudra ensuite écrire le code minimal qui servira à faire passer nos deux tests. Cela peut-être :

```
public static int Factorielle(int a)
{
    return 1;
}
```

Si nous exécutons nos tests, nous voyons que cette méthode est fonctionnelle car ils passent tous. La suite de la méthode consiste à refactoriser le code, à l'optimiser. Ici, il n'y a rien à faire, car c'est vraiment simple. On se rend compte par contre qu'on n'a pas couvert énormément de cas de tests ; faire juste des tests avec 0 et 1 c'est un peu léger. . . Nous savons que la factorielle de 2 vaut 2, la factorielle de 3 vaut 6, la factorielle de 4 vaut 24, etc. Continuons à écrire des tests (il faut bien sûr garder les anciens tests afin d'être sûrs qu'on couvre un maximum de cas) :

```
static void Main(string[] args)
{
    int valeur = 0;
    int resultat = Factorielle(valeur);
    if (resultat != 1)
        Console.WriteLine("Le test a raté");

    valeur = 1;
    resultat = Factorielle(valeur);
    if (resultat != 1)
        Console.WriteLine("Le test a raté");

    valeur = 2;
    resultat = Factorielle(valeur);
    if (resultat != 2)
        Console.WriteLine("Le test a raté");
```

```

    valeur = 3;
    resultat = Factorielle(valeur);
    if (resultat != 6)
        Console.WriteLine("Le test a raté");

    valeur = 4;
    resultat = Factorielle(valeur);
    if (resultat != 24)
        Console.WriteLine("Le test a raté");
}
```

Et nous pouvons écrire une méthode `Factorielle` qui fait passer ces tests :

```
public static int Factorielle(int a)
{
    if (a == 2)
        return 2;
    if (a == 3)
        return 6;
    if (a == 4)
        return 24;
    return 1;
}
```

Lançons les tests : nous voyons que tout est Ok. Cependant, nous n'allons pas faire des `if` en déclinant tous les cas possibles, il faut donc repasser par l'étape d'amélioration et de refactorisation du code, afin d'éviter les redondances de code, d'améliorer les algorithmes, etc. Cette opération devient sans risque puisque le test est là pour nous assurer que la modification que l'on vient de faire est sans régression, si le test passe toujours bien sûr… Nous voyons que nous pouvons améliorer le code en utilisant la vraie formule de la factorielle :

```
public static int Factorielle(int a)
{
    int total = 1;
    for (int i = 1 ; i <= a ; i ++)
    {
        total *= i;
    }
    return total;
}
```

Ce qui permet d'illustrer que, par exemple, la factorielle de 5 est égale à $1 \times 2 \times 3 \times 4 \times 5$. Relançons nos tests, ils passent tous. Parfait. Nous sommes donc certains que notre changement de code n'a pas altéré la fonctionnalité, car les tests continuent de passer. On peut même rajouter des tests pour le plaisir, comme la factorielle de 10, histoire d'avoir quelque chose d'un peu plus grand :

```
valeur = 10;
resultat = Factorielle(valeur);
```

```
if (resultat != 3628800)
    Console.WriteLine("Le test a raté");
```

Est-ce que cette méthode est optimisable ? Sûrement. Est-ce qu'il y a un risque à optimiser cette méthode ? Aucun ! En effet, nos tests nous garantissent que s'ils continuent à passer, alors une optimisation n'entraîne pas de régression dans la fonctionnalité. On sait par exemple qu'il y a un autre moyen pour calculer une factorielle. Par exemple, pour calculer la factorielle de 5, il suffit de multiplier 5 par la factorielle de 4. Pour calculer la factorielle de 4, il faut multiplier 4 par la factorielle de 3, et ainsi de suite jusqu'à arriver à 1... Bref, pour obtenir une factorielle on peut se servir du résultat de la factorielle du nombre précédent. Ce qui peut s'écrire :

```
public static int Factorielle(int a)
{
    if (a <= 1)
        return 1;
    return a * Factorielle(a - 1);
}
```

Ici la méthode `Factorielle` est une méthode récursive, c'est-à-dire qu'elle s'appelle elle-même. Cela nous permet d'indiquer que la factorielle d'un nombre correspond à ce nombre multiplié par la factorielle du nombre précédent. Bien sûr, il faut s'arrêter à un moment dans la récursion. On s'arrête ici quand on atteint le chiffre 1. Pour s'assurer que cette factorielle fonctionne bien, il suffit de relancer les tests. Tout est Ok, c'est parfait !

Voilà donc un exemple de TDD. Bien sûr, la méthode est ici poussée au maximum pour que vous compreniez l'intérêt de cette pratique. On peut gagner du temps en partant directement sur la bonne implémentation. Vous verrez qu'il y a toujours des premiers essais qui satisfont les tests mais qu'il sera possible d'améliorer régulièrement le code. Ceci devient possible grâce aux tests qui nous assurent que tout continue à bien fonctionner. La pratique du TDD dépend de la façon dont le développeur appréhende sa philosophie de développement. Elle est présentée ici pour sensibiliser ce dernier à cette pratique mais son utilisation n'est pas du tout obligatoire. Voilà pour les tests basiques. Cependant, utiliser une application console pour faire ses tests, ce n'est pas très pratique, vous en conviendrez. Nous avons besoin d'outils !

Le framework de test

Un framework de test est aux tests ce qu'un IDE est au développement. Il fournit un environnement structuré permettant l'exécution de tests, ainsi que des méthodes pour aider au développement de ceux-ci. Il existe plusieurs frameworks de test. Microsoft dispose de son framework, **mstest**, qui est disponible dans les versions payantes de Visual Studio. Son intérêt est qu'il est fortement intégré à l'IDE. Son défaut est qu'il ne fonctionne pas avec les versions gratuites de l'environnement de développement. Comme nous sommes partis dans ce livre avec la version gratuite, Visual C# Express, nous n'allons pas pouvoir utiliser mstest.

Par contre, il existe d'autres framework de test, gratuits, comme le très connu **NUnit**. NUnit est la version .NET du framework XUnit, qui se décline pour plusieurs environnements, avec par exemple PHPUnit pour le langage PHP, JUnit, pour java, etc. Première chose à faire : installer NUnit. Pour cela, utilisez le code web suivant :

▷ Télécharger NUnit
Code web : 481629

La version que j'utilise dans ce livre est la version 2.5.10. Démarrez l'installation, comme indiqué à la figure 38.1.

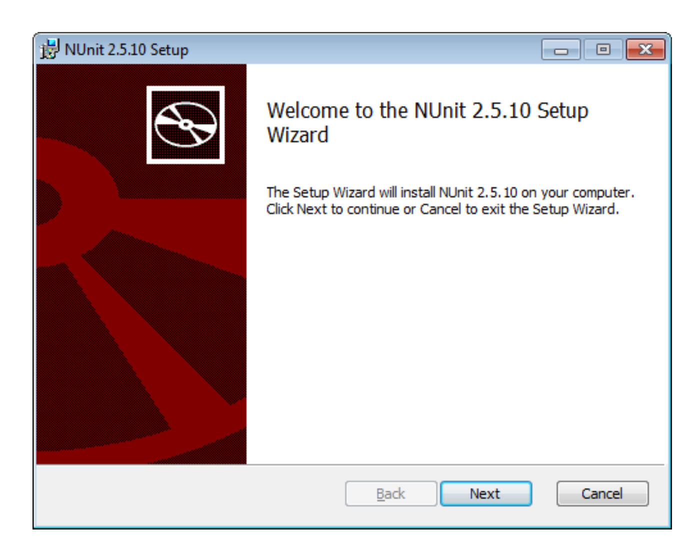


FIGURE 38.1 – Installation de NUnit

L'installation est en anglais, mais assez facile à suivre. Cliquez sur `Next` puis, après avoir accepté la licence, vous pouvez choisir l'installation classique (voir la figure 38.2).

Une fois le framework de test installé, nous pouvons créer un nouveau projet qui contiendra une fonctionnalité à tester. Je l'appelle `MaBibliothequeATester`. En général, nous allons surtout tester des assemblys avec NUnit. Je crée donc un projet de type bibliothèque de classes. Ce projet ne sera pas exécutable, car il ne s'agit pas d'une application console. À l'intérieur, je vais pouvoir créer une classe utilitaire, disons `Math`, qui contiendra notre méthode de calcul de factorielle :

```
public static class Math
{
    public static int Factorielle(int a)
    {
        if (a <= 1)
            return 1;
        return a * Factorielle(a - 1);
    }
}
```

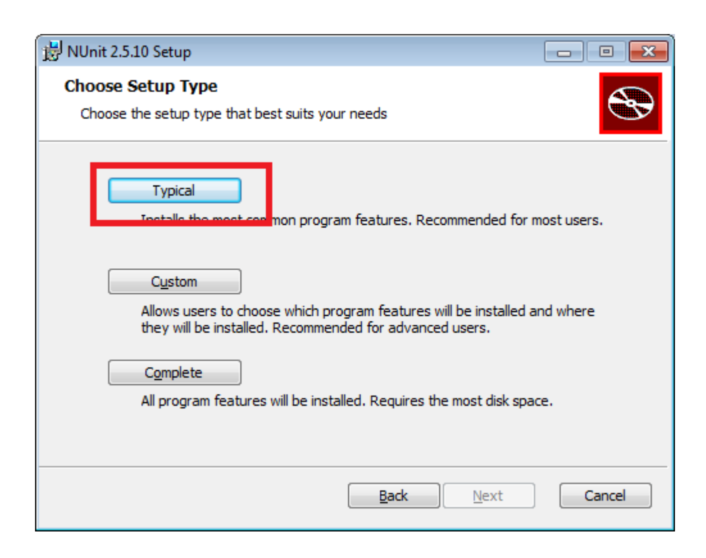


FIGURE 38.2 – Choix de l'installation classique

Ensuite, ajoutons un nouveau projet de type bibliothèque de classes où nous allons mettre nos tests unitaires, appelons-le `MathTests.Unit`. Ce n'est pas une norme absolue, mais je vous conseille de suffixer vos projets de test avec `.Unit`, afin de les identifier facilement.

Les tests doivent se mettre dans une classe spéciale. Ici aussi, pas de règle de nommage obligatoire, mais il est intéressant d'avoir une norme pour s'y retrouver facilement. Je vous propose de nommer les classes de tests en commençant par le nom de la classe que l'on doit tester, suivie du mot `Tests`. Ce qui donne : `MathTests`. Pour être reconnue par le framework de test, la classe doit respecter un certain nombre de contraintes. Elle doit dans un premier temps être décorée de l'attribut `[TestFixture]`. Il s'agit d'un attribut qui permet à NUnit de reconnaître les classes qui contiennent des tests.

Cet attribut étant dans une assembly de NUnit, vous devez rajouter une référence à l'assembly `nunit.framework`, ainsi qu'illustré à la figure 38.3.

Vous devez ensuite inclure l'espace de noms adéquat :

```
using NUnit.Framework;
```

Nous allons pouvoir créer des méthodes à l'intérieur de cette classe. De la même façon, une méthode pourra être reconnue comme une méthode de test si elle est décorée de l'attribut `[Test]`. Ici aussi, il est intéressant de suivre une règle de nommage afin de pouvoir identifier rapidement l'intention de la méthode de test. Je vous propose le nommage suivant : `MethodeTestee_EtatInitial_EtatAttendu()`. Par exemple, une méthode de test permettant de tester la factorielle pourrait s'appeler :

```
[TestFixture]
public class MathTests
{
    [Test]
```

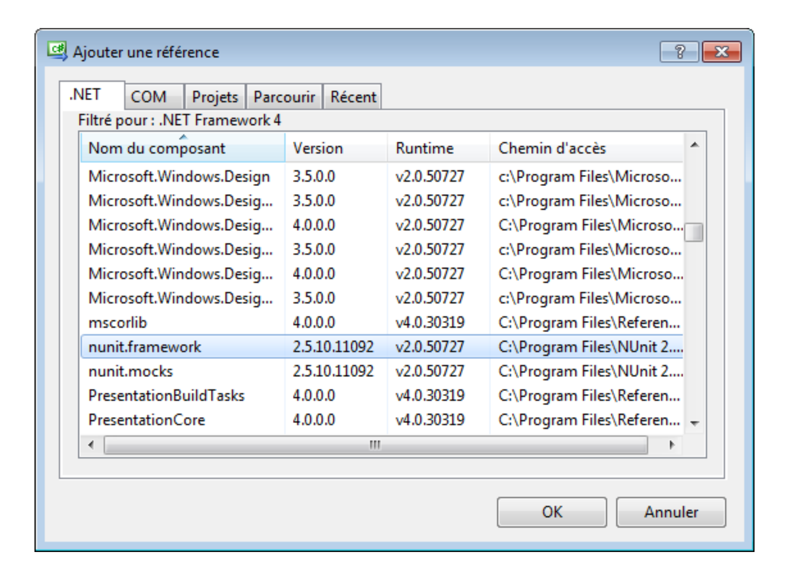


FIGURE 38.3 – Ajout d'une référence au framework de test

```
    public void Factorielle_AvecValeur3_Retourne6()
    {
        // test à faire
    }
}
```

Il existe plein d'autres attributs que vous découvrirez ultérieurement. Il est temps de passer à l'écriture du test et surtout à la vérification du résultat. Pour cela, on utilise des méthodes de NUnit qui nous permettent de vérifier par exemple qu'une valeur est égale à une autre attendue. Cela se fait grâce à la méthode `Assert.AreEqual()` :

```
[Test]
public void Factorielle_AvecValeur3_Retourne6()
{
    int valeur = 3;
    int resultat = MaBibliothequeATester.Math.Factorielle(
        valeur);
    Assert.AreEqual(6, resultat);
}
```

Elle permet de vérifier que la variable valeur vaut bien 6. Rajoutons tant qu'on y est une méthode de test qui échoue :

```
[Test]
public void Factorielle_AvecValeur10_Retourne1()
{
    int valeur = 10;
    int resultat = MaBibliothequeATester.Math.Factorielle(
        valeur);
```

```
    Assert.AreEqual(1, resultat, "La valeur doit être égale à 1");
}
```



Il est important que chaque méthode qui s'occupe de tester une fonctionnalité, le fasse à l'aide d'un cas unique comme illustré juste au-dessus. La première méthode teste la fonctionnalité `Factorielle` pour le cas où la valeur vaut 3 et la seconde s'occupe du cas où la valeur vaut 10. Vous pouvez rajouter autant de méthodes de tests que vous le souhaitez tant qu'elles sont décorées de l'attribut `[Test]`.

J'en ai profité pour ajouter un message qui permettra d'indiquer des informations complémentaires si le test échoue. Compilez maintenant le projet, allez dans le répertoire d'installation de NUnit (`C:\Program Files\NUnit 2.5.10\bin\net-2.0`) et lancez l'application `nunit.exe` (voir figure 38.4).

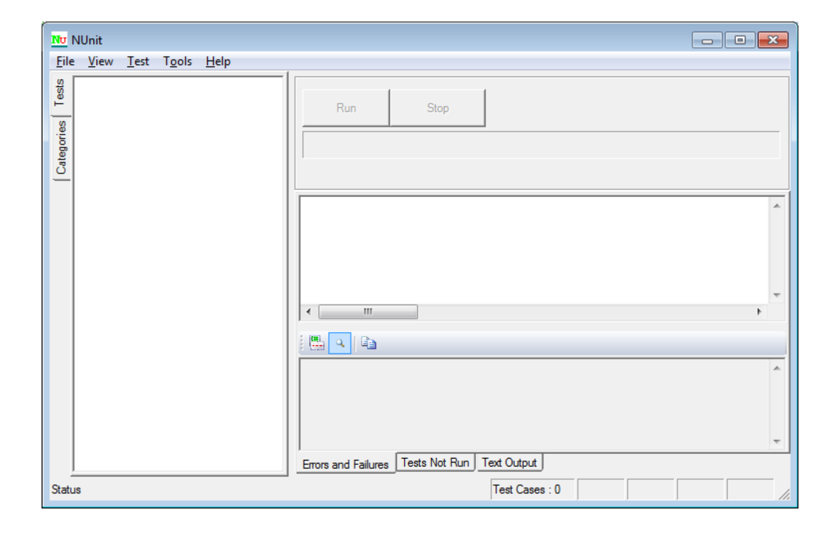


FIGURE 38.4 – Interface de NUnit

La première chose à faire est de créer un nouveau projet (voir la figure 38.5).

Appelez-le `ProjetTest` par exemple. Il faut ensuite ajouter une assembly de test ; allez dans `Project` > `Add Assembly` comme indiqué à la figure 38.6.

Pour finir, allez pointer l'assembly de tests, à savoir `MathTests.Unit.dll`. NUnit analyse l'assembly et fait apparaître la liste des tests qui composent notre assembly, en se basant sur les attributs `TestFixture` et `Test` (voir la figure 38.7).

Nous pouvons à présent lancer les tests en cliquant sur `Run` ! On s'aperçoit rapidement, en observant la figure 38.8, qu'il y a un test qui passe (icône verte) et un test qui échoue (icône rouge).

Forcément, notre test n'était pas bon, il faut le réécrire. Nous voyons également qu'il

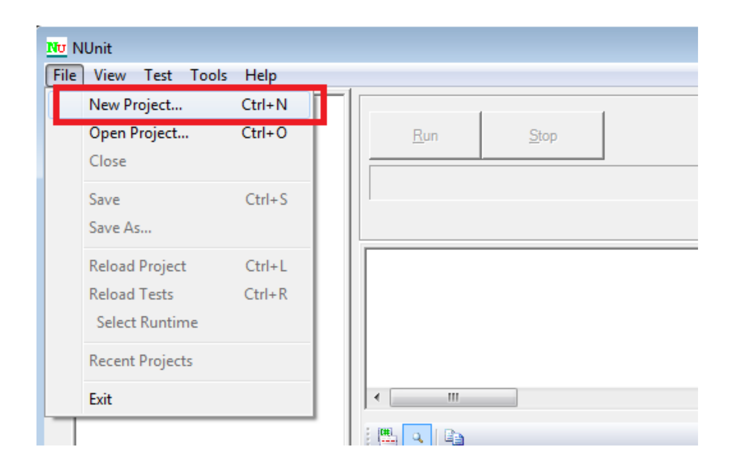


Figure 38.5 – Création d'un nouveau projet NUnit

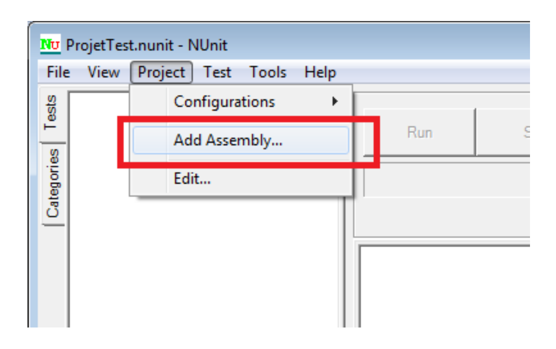


Figure 38.6 – Ajout de l'assembly contenant les tests

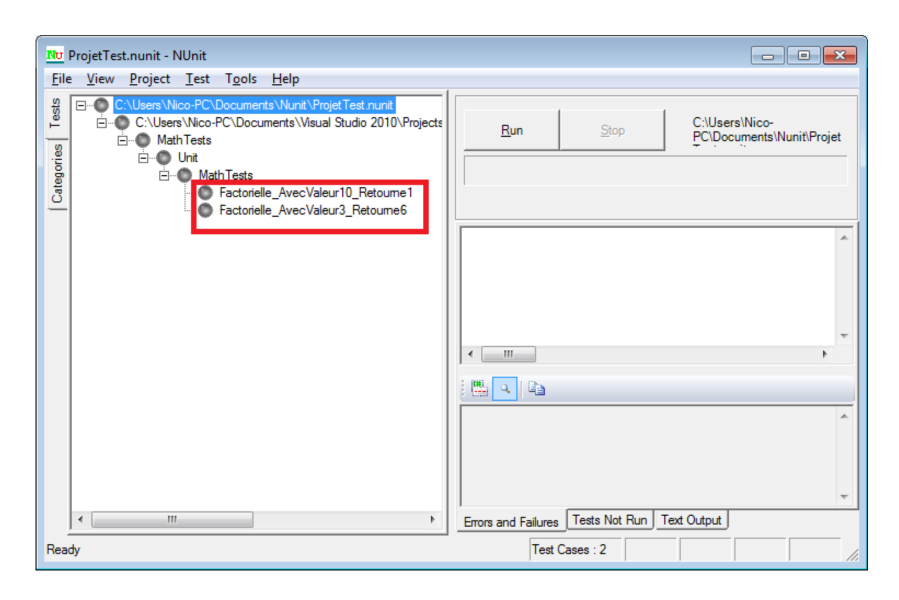


Figure 38.7 – Les tests présents dans l'assembly

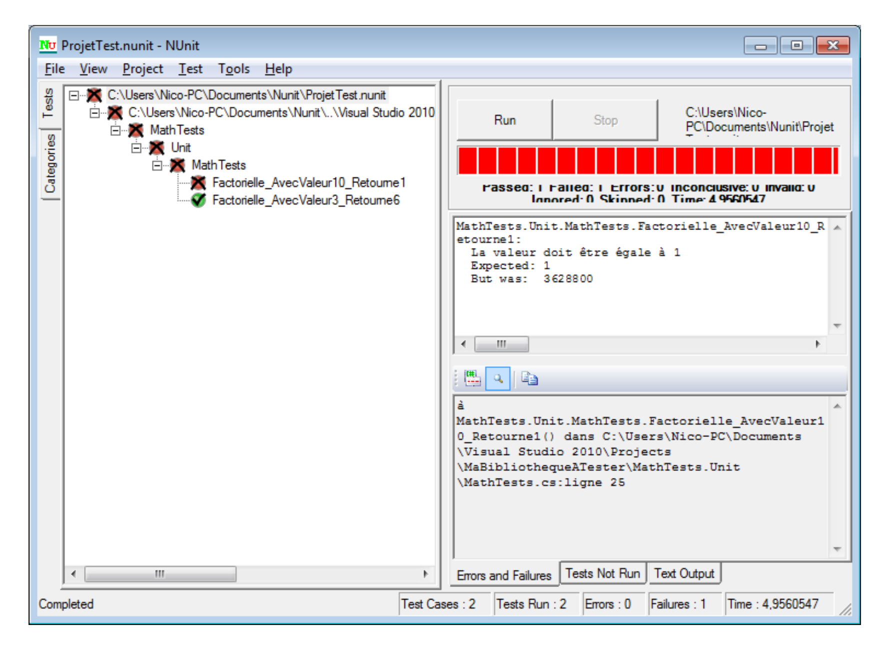


Figure 38.8 – Première exécution des tests

nous indique que le résultat attendu était 1 alors que le résultat obtenu est de 3628800. Nous pouvons également voir le message que nous avons demandé d'afficher en cas d'erreur. Le souci avec NUnit, c'est qu'à partir du moment où il a chargé la `dll` pour lancer les tests, il n'est plus possible de faire de modifications, car toute tentative de compilation provoquera une erreur où il sera mentionné qu'il ne peut pas faire de modifications car le fichier est déjà utilisé ailleurs. Ce qui est vrai. Nous serons donc obligés de fermer puis de rouvrir NUnit. À noter que dans les versions payantes de Visual Studio, nous avons la possibilité de configurer NUnit en tant qu'outil externe, ce que nous ne pouvons pas faire avec la version gratuite. Il va falloir faire avec... C'est un des inconvénients de la gratuité... ! Nous pouvons cependant un peu tricher en définissant un événement de post-compilation, qui consiste à lancer NUnit automatiquement. Pour cela, allez dans les propriétés du projet, onglet `Événements de build` et tapez la commande suivante : `"C:\Program Files\NUnit 2.5.10\bin\net-2.0\nunit.exe"` $(TargetFileName). Ici, nous indiquons qu'après la compilation, il va lancer le programme `nunit.exe` en prenant en paramètre le résultat de la compilation, représenté par la variable interne de Visual C# Express : « `$(TargetFileName)` » (voir la figure 38.9).

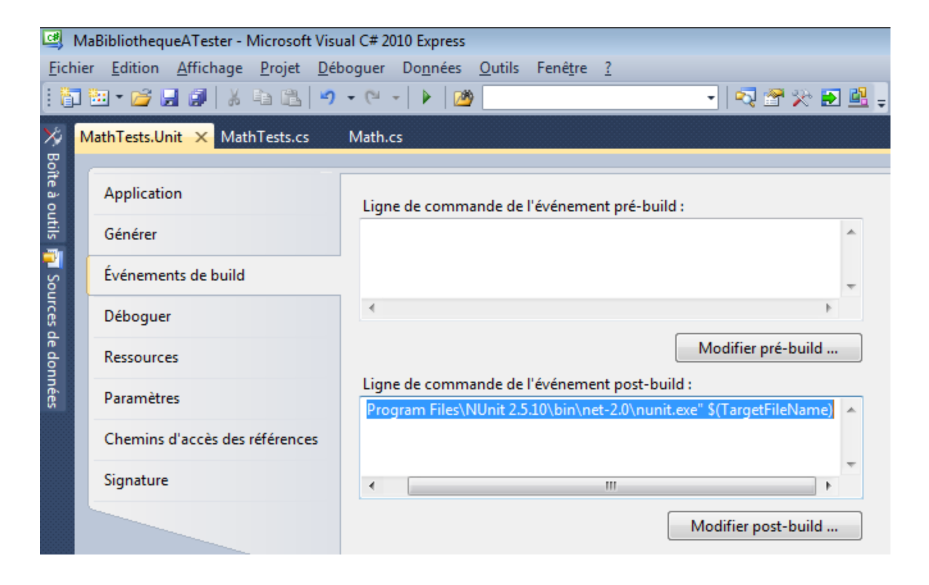


FIGURE 38.9 – Modification des éléments de post-compilation

Par contre, cela veut dire que NUnit va se lancer à chaque compilation, ce qui n'est peut-être pas le but recherché... Il faudra également fermer NUnit avant de pouvoir faire quoi que ce soit d'autre. À noter que maintenant que nous savons faire de l'introspection sur les méthodes et les attributs d'une classe, nous devrions être capables de créer une petite application qui exécute les tests automatiquement ! Pour en finir avec NUnit, notons qu'il y a beaucoup de méthodes permettant de vérifier si un résultat est correct. Regardons les assertions suivantes :

```
bool b = true;
```

```
Assert.IsTrue(b);
string s = null;
Assert.IsNull(s);
int i = 10;
Assert.Greater(i, 6);
```

Elles parlent d'elles-mêmes. La première permet de vérifier qu'une condition est vraie. La deuxième permet de vérifier la nullité d'une variable. La dernière permet de vérifier qu'une variable est bien supérieure à une autre. À noter qu'elles ont chacune leur pendant (`IsFalse`, `IsNotNull`, `Less`). En regardant la complétion automatique, vous découvrirez d'autres méthodes de vérification, mais celles-ci sont globalement suffisantes. Nous pouvons également utiliser une syntaxe un peu plus parlante, surtout pour les anglophones :

```
Assert.That(i, Is.EqualTo(10));
```

Il est également possible d'utiliser un attribut pour vérifier qu'une méthode lève bien une exception, par exemple :

```
[Test]
[ExpectedException(typeof(FormatException))]
public void ToInt32_AvecChaineNonNumerique_LeveUneException()
{
    Convert.ToInt32("abc");
}
```

Dans ce cas, le test passe si la méthode lève bien une `FormatException`.

Avant de terminer, présentons deux attributs supplémentaires : les attributs `SetUp` et `TearDown`. Ils permettent de décorer des méthodes qui seront appelées respectivement avant et après chaque test. C'est l'endroit idéal pour factoriser des initialisations ou des nettoyages dont dépendent tous les tests.

```
[TestFixture]
public class MathTests
{
    [SetUp]
    public void InitialisationDesTests()
    {
        // rajouter les initialisations
    }

    [Test]
    public void Factorielle_AvecValeur3_Retourne6()
    {
        // test à faire
    }

    [TearDown]
    public void NettoyageDesTests()
    {
```

```
            // nettoyer les variables, ...
        }
}
```

Il existe plein d'autres choses utiles à dire sur NUnit, comme la description des autres attributs, mais il n'est pas utile de les voir toutes à notre niveau. N'hésitez pas à aller voir sur internet des informations plus poussées si vous ressentez le besoin d'approfondir votre maîtrise des tests.

Le framework de simulacre

Un framework de simulacre fournit un moyen de tester une méthode en l'isolant du reste du système. Imaginons par exemple une méthode qui permette de récupérer la météo du jour, en allant la lire dans une base de données. Nous avons ici un problème, car lorsque nous exécutons le test le lundi, il pleut. Quand nous exécutons le test le mardi, il fait beau, etc. Nous avons ici une information qui varie au cours du temps. Il est donc difficile de tester automatiquement que la méthode arrive bien à construire la météo du jour à partir de ces informations, vu qu'elles varient. Le but de ces frameworks est de pouvoir bouchonner le code dont notre développement dépend, afin de pouvoir le tester unitairement, sans dépendance et isolé du reste du système. Cela veut dire que dans notre test, nous allons remplacer la lecture en base de données par une fausse méthode qui renvoie toujours qu'il fait beau. Cependant, ceci doit se faire sans modifier notre application, sinon cela n'a pas d'intérêt. Voilà à quoi servent ces framework de simulacres. Il en existe plusieurs, plus ou moins complexes. Citons par exemple **Moq** (prononcez « moque-you ») ou encore **Moles** (il y en a plein d'autres). L'intérêt de Moq est qu'il est simple d'accès, nous allons le présenter rapidement. Il permet de faire des choses simples et facilement. Tandis que Moles est un peu plus évolué mais plus complexe à prendre en main. Vous y reviendrez sans doute ultérieurement. Pour le télécharger, utilisez le code web suivant :

▷ Télécharger Moq
Code web : 225970

Pas de système d'installation évolué, il y aura juste une `dll` à référencer. Ajoutez donc la référence à la `dll moq.dll` qui se trouve dans le sous-répertoire `NET40`. Ensuite, pour pouvoir bouchonner facilement une classe, elle doit implémenter une interface. Imaginons la classe d'accès aux données suivante :

```
public class Dal : IDal
{
    public Meteo ObtenirLaMeteoDuJour()
    {
        // ici, c'est le code pour lire en base de données
        // mais finalement, peu importe ce qu'on y met vu qu'on
             va bouchonner la méthode
        throw new NotImplementedException();
    }
}
```

Qui implémente l'interface suivante :

```
public interface IDal
{
    Meteo ObtenirLaMeteoDuJour();
}
```

Avec l'objet `Meteo` suivant :

```
public class Meteo
{
    public double Temperature { get; set; }
    public Temps Temps { get; set; }
}
```

Et l'énumération `Temps` suivante :

```
public enum Temps
{
    Soleil,
    Pluie
}
```

Comme nous allons le voir, nous pouvons également écrire un test qui bouchonne l'appel à la méthode `ObtenirLaMeteoDuJour`, qui doit normalement aller lire en base de données, pour renvoyer un objet à la place. Pour bien montrer ce fonctionnement, j'ai fait en sorte que la méthode lève une exception, comme ça, si on passe dedans ça sera tout de suite visible. La méthode de test classique devrait être :

```
[Test]
public void ObtenirLaMeteoDuJour_AvecUnBouchon_RetourneSoleil()
{
    IDal dal = new Dal();
    Meteo meteoDuJour = dal.ObtenirLaMeteoDuJour();
    Assert.AreEqual(25, meteoDuJour.Temperature);
    Assert.AreEqual(Temps.Soleil, meteoDuJour.Temps);
}
```

Si nous exécutons le test, nous aurons une exception. Utilisons maintenant Moq pour bouchonner cet appel et le remplacer par ce que l'on veut :

```
[Test]
public void ObtenirLaMeteoDuJour_AvecUnBouchon_RetourneSoleil()
{
    Meteo fausseMeteo = new Meteo { Temperature = 25, Temps =
        Temps.Soleil };
    Mock<IDal> mock = new Mock<IDal>();
    mock.Setup(dal => dal.ObtenirLaMeteoDuJour()).Returns(
        fausseMeteo);

    IDal fausseDal = mock.Object;
    Meteo meteoDuJour = fausseDal.ObtenirLaMeteoDuJour();
```

```
    Assert.AreEqual(25, meteoDuJour.Temperature);
    Assert.AreEqual(Temps.Soleil, meteoDuJour.Temps);
}
```

On utilise l'objet générique `Mock` pour créer un faux objet du type de notre interface. On utilise la méthode `Setup` à travers une expression lambda pour indiquer que la méthode `ObtenirLaMeteoDuJour` retournera en fait un faux objet météo. Cela se fait tout naturellement en utilisant la méthode `Returns()`. L'avantage de ces constructions est que la syntaxe parle d'elle-même à partir du moment où l'on connaît les expressions lambdas. On obtient ensuite une instance de notre objet grâce à la propriété `Object` et c'est ce faux objet que nous pourrons comparer à nos valeurs. Bien sûr, ici, ce test n'a pas grand intérêt. Mais il faut le voir à un niveau plus général. Imaginons que nous ayons besoin de tester la fonctionnalité qui met en forme cet objet météo récupéré de la base de données ou bien l'algorithme qui nous permet de faire des statistiques sur ces données météos... Là, nous sommes sûrs de pouvoir nous baser sur une valeur connue de la dépendance à la base de données. Cela permettra également de décliner tous les cas possibles en changeant la valeur du bouchon et de faire les tests les plus exhaustifs possibles. Nous pouvons faire la même chose avec les propriétés. Imaginons la classe suivante dont la propriété valeur retourne un nombre aléatoire :

```
public interface IGenerateur
{
    int Valeur { get; }
}

public class Generateur : IGenerateur
{
    private Random r;
    public Generateur()
    {
        r = new Random();
    }

    public int Valeur
    {
        get
        {
            return r.Next(0, 100);
        }
    }
}
```

Nous pourrions avoir besoin de bouchonner cette propriété pour qu'elle renvoie un nombre connu à l'avance. Cela se fera de la même façon :

```
Mock<IGenerateur> mock = new Mock<IGenerateur>();
mock.Setup(generateur => generateur.Valeur).Returns(5);

Assert.AreEqual(5, mock.Object.Valeur);
```

Ici, la propriété `Valeur` renverra toujours 5 en se moquant bien du générateur de nombre aléatoire... Je m'arrête là pour l'aperçu de ce framework de simulacre. Nous avons pu voir qu'il pouvait facilement bouchonner des dépendances nous permettant de faciliter la mise en place de nos tests unitaires. Rappelez-vous que pour qu'un test soit efficace, il doit pouvoir se concentrer sur un point précis du code sans être gêné par les dépendances éventuelles qui peuvent agir sur l'état du test à un instant « t ».

En résumé

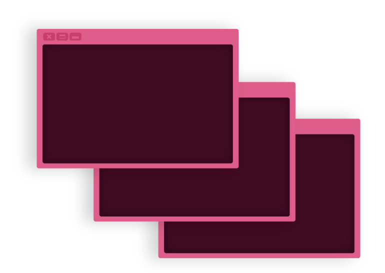
– Les tests unitaires sont un moyen efficace de tester des bouts de code dans une application afin de garantir son bon fonctionnement.
– Ils sont un filet de sécurité permettant de faire des opérations de maintenance, de refactoring ou d'optimisation sur le code.
– Les frameworks de tests unitaires sont en général accompagnés d'outils permettant de superviser le bon déroulement des tests et la couverture de tests.

Chapitre 39

Les types d'applications pouvant être développées en C#

Difficulté :

Vous savez quoi ? Avec le C# on peut créer autre chose que des applications console ! On peut faire des applications avec des boutons et des menus, ou des sites web et même des jeux ! Dans ce chapitre, je vais vous indiquer rapidement les différents types d'applications qu'on peut développer avec le C#. Chaque paragraphe de ce chapitre nécessiterait un livre entier pour être correctement traité, aussi, ce ne sera qu'un très bref aperçu. J'espère que vous me pardonnerez d'aller si vite mais je souhaite néanmoins piquer votre curiosité pour vous donner envie d'aller explorer tous ces nouveaux mondes qui s'ouvrent à vous ! Il y aura certainement des notions que vous ne comprendrez pas complètement en fonction des thèmes abordés. Il faudra aller jeter un œil complémentaire sur internet ou attendre un prochain livre !



Créer une application Windows avec WPF

Les applications Windows sont ce qu'on appelle des applications clients lourds. Elles s'exécutent directement depuis notre système d'exploitation. Nous pouvons créer toutes sortes d'applications, comme un traitement de texte, une calculatrice, etc.

Nous avons déjà créé une application Windows à travers notre projet console, sauf que nous étions rapidement limités. Avec WPF, nous allons pouvoir créer des applications graphiques avec des boutons, des menus, etc. Bref, tout ce qui compose une application habituelle. WPF signifie *Windows Presentation Foundation*. Il s'agit d'une bibliothèque permettant de réaliser des applications graphiques. Ces applications sont dites événementielles car elles réagissent à des événements (clic sur un bouton, redimensionnement de la fenêtre, saisie de texte, etc.)

Pour construire des applications WPF, nous aurons besoin de deux langages. Un langage de présentation qui va permettre de décrire le contenu de notre fenêtre : le **XAML**(prononcez « xamelle ») et du C# qui va permettre de faire tout le code métier.

Créer une application web avec ASP.NET

ASP.NET c'est la plateforme de Microsoft pour réaliser des applications web. C'est un peu comme PHP, sauf que, vous vous en doutez, ASP.NET s'appuie massivement sur le framework .NET. Et tout comme WPF, il s'agit de bibliothèques qui vont permettre de réaliser facilement son site web.

Il existe deux ASP.NET : l'**ASP.NET WebForms** et l'**ASP.NET MVC**. ASP.NET WebForms c'est tout un mécanisme qui permet de faciliter la création d'une application web en faisant comme si c'était une application Windows. C'est-à-dire que le framework s'occupe de gérer toute la persistance d'informations entre les différents états des pages. Il permet aussi de travailler avec une approche événementielle, comme une application Windows. Le premier but d'ASP.NET WebForms était de faire en sorte que les personnes qui avaient déjà fait du développement Windows (avec des langages comme le Visual Basic ou autre) puissent facilement faire du développement web, dans un contexte qui leur serait familier.

ASP.NET MVC est plus récent et offre une approche où le développeur doit bien connaître tous les mécanismes du web. Il offre également une plus grande maîtrise sur le rendu du site web. Enfin, il intègre par défaut tous les mécanismes éprouvés du fameux patron de conception (design pattern) MVC. On ne peut pas dire qu'ASP.NET WebForms soit mieux ou moins bien qu'ASP.NET MVC. Il s'agit de deux façons différentes de créer des sites web. Chacune a ses avantages et ses inconvénients. Par contre, les deux se basent sur un socle commun qui est le cœur d'ASP.NET.

Créer une application client riche avec Silverlight

Nous avons vu qu'il existait des applications type clients lourds, comme les applications console ou les applications WPF. Nous avons également vu des applications web, avec ASP. NET WebForms ou ASP.NET MVC. Il existe quelque chose entre les deux, ce sont les applications dites « client riche ». Ce sont des applications qui ressemblent à des applications lourdes, mais qui s'exécutent à l'intérieur d'un navigateur internet plutôt que directement au niveau du système d'exploitation. Vous connaissez sûrement le célèbre « flash », très populaire grâce à la multitude de jeux disponibles sur internet. Microsoft possède également des bibliothèques permettant de réaliser des applications clients riches : **Silverlight**.

Une application client riche s'exécute donc directement dans un navigateur internet, comme *Internet Explorer*, *Firefox* ou *Chrome*. Ces applications s'exécutent dans un plugin du navigateur. Pour exécuter des applications flash ou des applications Silverlight, le navigateur devra posséder le plugin adéquat. Du fait qu'elles s'exécutent dans un navigateur, ces applications ont quelques restrictions. Elles ne peuvent par défaut pas accéder au contenu du disque dur de l'utilisateur, ce qui est finalement plutôt pas mal pour une application disponible directement sur internet. Elles s'exécutent uniquement dans la zone mémoire du navigateur, une espèce de bac à sable dont on ne peut pas s'échapper et d'où il est impossible d'accéder aux ressources directes de l'ordinateur sur lequel s'exécute l'application, au contraire des applications WPF par exemple. Ces applications clients riches ressemblent énormément aux applications clients lourds avec quelques restrictions. Silverlight est donc une espèce de WPF allégé qui ne garde que l'essentiel de l'essentiel. Nous serons donc capables de réaliser des applications s'exécutant dans notre navigateur grâce au C# et au XAML.

Le graphisme et les jeux avec XNA

Eh oui, il est aussi possible de réaliser des jeux avec le C#. Même si tout ce qu'on a vu auparavant permet la réalisation d'applications de gestion, Microsoft dispose aussi de tout ce qu'il faut pour réaliser des jeux, grâce à XNA. XNA est un ensemble de bibliothèques permettant de créer des applications graphiques mais c'est également un outil permettant d'intégrer des contenus facilement dans des jeux (comme des images ou des musiques). Avec XNA, il est possible de faire des jeux qui fonctionnent sous Windows mais également sous Xbox ou sur les téléphones Windows.

Créer une application mobile avec Windows Phone 7

Le C# nous permet également de développer des applications pour téléphones mobiles équipés du système d'exploitation Windows Phone. C'est un point très important car il est possible de mutualiser beaucoup de choses que nous avons apprises pour les transposer dans le monde en plein essor des smartphones.

Ce qui est intéressant avec les applications Windows Phone c'est que nous réutilisons

le savoir que nous avons pu acquérir dans les autres types d'applications C#. En effet, pour réaliser des applications de gestion, nous allons utiliser Silverlight et pour réaliser des jeux, nous utiliserons XNA.

Créer un service web avec WCF

Avec le C# il est également très facile de créer des services web. Un service web permet en général d'accéder à des fonctionnalités depuis n'importe où, à travers internet. Citons par exemple les services web d'Amazon qui nous permettent de récupérer des informations sur des livres, ou encore des services web qui permettent d'obtenir la météo du jour. Bref, c'est un moyen de communication entre applications hétérogènes potentiellement situées à des emplacements physiques très éloignés. En imaginant que nous ayons également besoin d'exposer des méthodes à l'extérieur, pour qu'un fournisseur vienne consulter l'état de nos commandes ou qu'un client puisse suivre l'avancée de la sienne. . ., nous allons devoir créer un service web. Le framework .NET dispose de tout un framework pour cela qui s'appelle WCF : *Windows Communication Foundation*.

Un service web est une espèce d'application web qui répond à des requêtes permettant d'appeler une méthode avec des paramètres et de recevoir en réponse le retour de la méthode. L'intérêt d'un service web est qu'il est indépendant de la technologie. Même si on écrit un service web avec du C#, il peut être appelé par du java ou du PHP.

Retrouvez des explications plus complètes et des exemples pratiques via le code web suivant :

▷ Aller plus loin
Code web : 777901

Pour conclure

Ça y est, ce livre est terminé.

Le C# n'a (presque) plus de secret pour nous. Tout au long de ce livre nous avons découvert comment développer des applications avec le C#. Il a fallu dans un premier temps apprendre les bases du C#, ce qui n'est pas sans douleurs quand on n'a jamais fait de programmation ! Pouvoir appréhender correctement ce qu'est une variable ou comment dérouler son premier programme n'est pas une mince affaire. Mais petit à petit, nous avons relevé le défi en présentant des fonctionnalités qui nous serviront tout le temps, pour la moindre petite application.

Forts de ces nouveaux apprentissages, nous avons ensuite découvert le monde de la programmation orientée objet et comment le C# était un vrai langage orienté objet. À travers des notions comme les classes et autres propriétés, nous avons pu tirer parti de cette façon de modéliser des applications pour les adapter à nos besoins. Je le reconnais, ce n'est pas une partie facile. Les différents concepts peuvent donner rapidement mal à la tête ! Ce qui est certain c'est que petit à petit, ils vont devenir de plus en plus clairs. On peut très bien commencer à développer en orienté objet sans vraiment en maîtriser toutes les subtilités. C'est en pratiquant et en restant curieux sur le sujet que vous progresserez.

Enfin, nous avons poussé un peu plus loin dans les arcanes du C#, nous permettant de réaliser des applications de plus en plus compliquées. Nous avons également montré comment utiliser les bases de données avec Entity Framework. Savoir lire et écrire dans une base de données est un élément fondamental pour toute application de gestion qui se respecte. Nous avons également aperçu ce que l'on pouvait faire avec le C#. De l'application Windows, en passant par les jeux ou les applications web, le C# combiné au framework .NET permet vraiment de faire beaucoup de choses !

Vous avez désormais les clés pour démarrer. Il ne reste plus qu'à vous plonger dans les domaines qui vous intéressent afin de réaliser les applications dont vous avez envie. Pourquoi pas une application pour les smartphones ? C'est très à la mode.

N'oubliez pas que c'est à force de pratiquer, d'essayer de créer des petites applications de rien du tout, que vous finirez par être un développeur confirmé, capable à son tour d'aider les autres. N'hésitez pas à faire et à refaire les TP. Après avoir fini le livre, vous pourrez sûrement améliorer vos premières versions des TP. Créez-en, fixez vous des petits défis et venez les échanger avec nous sur le Site du Zéro ! ;-)

Index

Dépôt légal : mars 2012
ISBN : 978-2-9535278-6-5
Code éditeur : 978-2-9535278
Imprimé en France

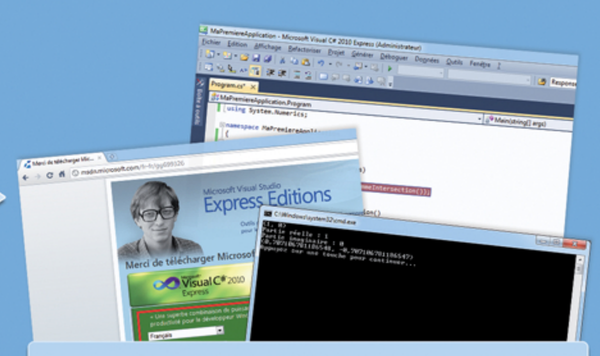
Achevé d'imprimer le 20 mars 2012
sur les presses de Corlet Imprimeur (Condé-sur-Noireau)
Numéro imprimeur : 144485



Mentions légales :
Conception couverture : Fan Jiyong
Illustrations chapitres : Fan Jiyong

APPRENEZ À DÉVELOPPER EN C#

Vous désirez **apprendre le C#** mais vous n'avez aucune connaissance en programmation ? Cet ouvrage est fait pour vous ! Grâce à ce livre **conçu pour les débutants**, découvrez pas à pas le **langage phare de Microsoft**, manipulez le framework .NET, apprenez la programmation orientée objet… et bien d'autres choses encore !



Plus de **30 chapitres** de difficulté progressive
Des **exercices réguliers** sous forme de TP
Un livre **entièrement en couleur**

Un cours conçu pour les débutants

- Le seul pré-requis est de savoir allumer son ordinateur
- Des exemples clairs et une pédagogie adaptée aux débutants
- Aucune connaissance en programmation n'est requise

La programmation en C# pas à pas

- Qu'est-ce que la programmation ? Quel langage choisir ? Pourquoi C# ?
- Installez Visual C# 2010 Express et écrivez votre premier programme
- Apprenez à manipuler les variables, les méthodes, les boucles…
- Initiez-vous aux concepts de la programmation orientée objet
- Pratiquez grâce aux TP : créez un jeu, une application de gestion bancaire, un simulateur de météo…
- Connectez-vous aux bases de données avec Entity Framework
- Découvrez les applications que l'on peut développer en C#

À qui ce livre est-il destiné ?

- Aux passionnés d'informatique qui veulent aller plus loin avec leur ordinateur
- Aux étudiants dans le domaine des nouvelles technologies qui recherchent un support de cours
- À toutes les personnes qui ont besoin de se former ou de se convertir à la programmation

À propos de l'auteur



Nicolas Hilaire

Passionné de développement depuis tout petit, Nicolas Hilaire est aujourd'hui ingénieur en informatique pour une grande entreprise d'e-commerce française.

Suite à plusieurs publications sur le C++/CLI, il a obtenu en 2007 le titre de *Microsoft Valuable Professional Visual C++*. Il partage depuis sa passion pour le C# et ASP.NET et obtient le titre *Microsoft Valuable Professional* ASP.NET en 2010. Curieux des autres langages, il continue à s'intéresser aux nouvelles technologies qui pointent le bout de leur nez.

Ce livre est issu du Site du Zéro

Retrouvez dans ce livre les cours du Site du Zéro dans une édition revue et corrigée avec un nouveau chapitre inédit du même auteur !

Téléchargez les codes source en ligne grâce aux « codes web » inclus dans ce livre.

ISBN : 978-2-9535278-6-5



Prix public : 33 € TTC



www.siteduzero.com

